

# pyAgrum Documentation

*Release 1.3.1*

**Pierre-Henri Wuillemin (Sphinx)**

**August 28, 2022**



# 1-FUNDAMENTAL COMPONENTS

<b>1</b>	<b>Reference manual</b>	<b>3</b>
1.1	Graphs manipulation . . . . .	3
1.2	Random Variables . . . . .	25
1.3	Potential and Instantiation . . . . .	45
1.4	Bayesian network . . . . .	63
1.5	Influence Diagram . . . . .	190
1.6	Credal Network . . . . .	203
1.7	Markov Network . . . . .	216
1.8	Probabilistic Relational Models . . . . .	230
1.9	pyAgrum.causal documentation . . . . .	235
1.10	pyAgrum.skbn documentation . . . . .	254
1.11	pyAgrum.lib.notebook . . . . .	261
1.12	pyAgrum.lib.image . . . . .	269
1.13	pyAgrum.lib.explain . . . . .	270
1.14	pyAgrum.lib.dynamicBN . . . . .	273
1.15	other pyAgrum.lib modules . . . . .	275
1.16	Functions from pyAgrum . . . . .	278
1.17	Other functions from aGrUM . . . . .	284
1.18	Exceptions from aGrUM . . . . .	286
1.19	Configuration for pyAgrum . . . . .	292
1.20	Tutorials on pyAgrum . . . . .	294
1.21	Inference in Bayesian networks . . . . .	311
1.22	Learning Bayesian networks . . . . .	330
1.23	Different Graphical Models . . . . .	363
1.24	Bayesian networks as scikit-learn compliant classifiers . . . . .	389
1.25	Causal Bayesian Networks . . . . .	408
1.26	Examples . . . . .	432
1.27	pyAgrum's specific features . . . . .	469
<b>2</b>	<b>Indices and tables</b>	<b>503</b>
	<b>Python Module Index</b>	<b>505</b>
	<b>Index</b>	<b>507</b>

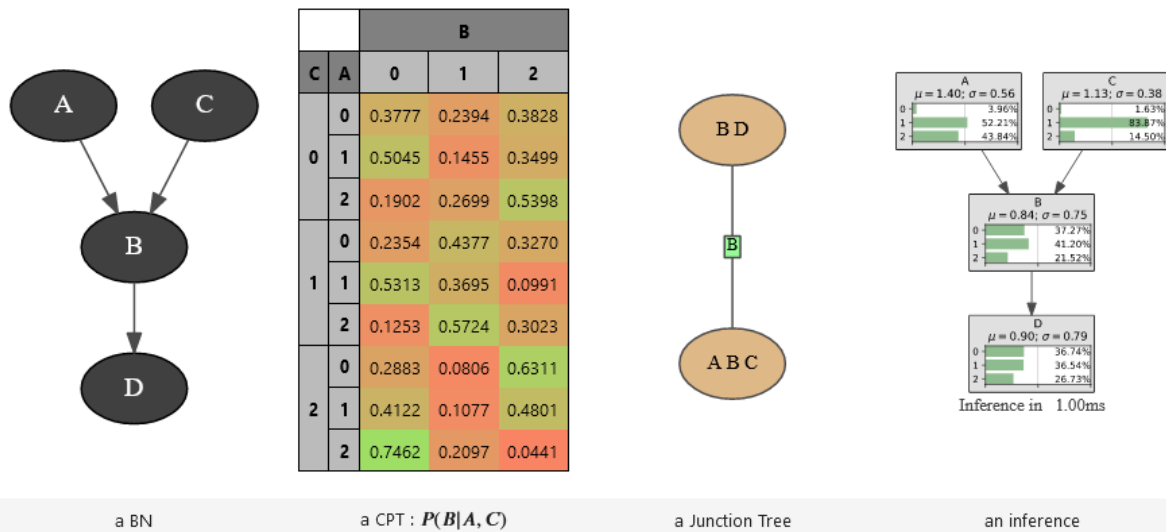


**pyAgrum** (<http://agrum.org>) is a scientific C++ and Python library dedicated to Bayesian networks (BN) and other Probabilistic Graphical Models. Based on the C++ **aGrUM** (<https://agrum.lip6.fr>) library, it provides a high-level interface to the C++ part of aGrUM allowing to create, manage and perform efficient computations with Bayesian networks and others probabilistic graphical models : Markov networks (MN), influence diagrams (ID) and LIMIDs, credal networks (CN), dynamic BN (dbn), probabilistic relational models (PRM).



(<http://agrum.org>)

```
bn=gum.fastBN("A->B;C->B->D",3)
gmb.sideBySide(bn,
    bn.cpt("B"),
    gmb.getJunctionTree(bn),
    gmb.getInference(bn),
    captions=['a BN', 'a CPT : $P(B|A,C)$', 'a Junction Tree', 'an inference'])
```



The module is generated using the **SWIG** (<http://www.swig.org>) interface generator. Custom-written code was added to make the interface more user friendly.

pyAgrum aims to allow to easily use (as well as to prototype new algorithms on) Bayesian network and other graphical models.

**pyAgrum contains :**

- a *comprehensive API documentation* (page 3).
- *tutorials as jupyter notebooks* (page ??).
- a *gitlab repository* (<https://gitlab.com/agrumery/aGrUM>).
- and a *website* (<http://agrum.org>).



## REFERENCE MANUAL

### 1.1 Graphs manipulation

In aGrUM, graphs are undirected (using edges), directed (using arcs) or mixed (using both arcs and edges). Some other types of graphs are described below. Edges and arcs are represented by pairs of int (nodeId), but these pairs are considered as unordered for edges whereas they are ordered for arcs.

For all types of graphs, nodes are int. If a graph of objects is needed (like [pyAgrum.BayesNet](#) (page 64)), the objects are mapped to nodeIds.

#### 1.1.1 Edges and Arcs

##### Arc

**class** pyAgrum.Arc(\*args)

pyAgrum.Arc is the representation of an arc between two nodes represented by int : the head and the tail.

**Arc(tail, head) -> Arc**

**Parameters:**

- **tail** (int) – the tail
- **head** (int) – the head

**Arc(src) -> Arc**

**Parameters:**

- **src** (Arc) – the pyAgrum.Arc to copy

**first()**

**Returns**

the nodeId of the first node of the arc (the tail)

**Return type**

int

**head()**

**Returns**

the id of the head node

**Return type**

int

**other(id)**

**Parameters**

**id** (int) – the nodeId of the head or the tail

**Returns**

the nodeId of the other node

**Return type**

int

**second()****Returns**

the nodeId of the second node of the arc (the head)

**Return type**

int

**tail()****Returns**

the id of the tail node

**Return type**

int

## Edge

**class** pyAgrum.**Edge**(\*args)

pyAgrum.Edge is the representation of an arc between two nodes represented by int : the first and the second.

**Edge(aN1,aN2) -> Edge****Parameters:**

- **aN1** (int) – the nodeId of the first node
- **aN2** (int) – the nodeId of the secondnode

**Edge(src) -> Edge****Parameters:**

- **src** (pyAgrum.Edge) – the Edge to copy

**first()****Returns**

the nodeId of the first node of the arc (the tail)

**Return type**

int

**other(id)****Parameters****id** (int) – the nodeId of one of the nodes of the Edge**Returns**

the nodeId of the other node

**Return type**

int

**second()****Returns**

the nodeId of the second node of the arc (the head)

**Return type**

int



## 1.1.2 Directed Graphs

### Digraph

**class** pyAgrum.DiGraph(\*args)

DiGraph represents a Directed Graph.

**DiGraph()** -> **DiGraph**

default constructor

**DiGraph(src)** -> **DiGraph**

**Parameters:**

- **src** (*pyAgrum.DiGraph*) – the digraph to copy

**addArc**(\*args)

Add an arc from tail to head.

**Parameters**

- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

**Raises**

- *pyAgrum.InvalidNode* (page 289) –
- **If head or tail does not belong to the graph nodes.** –

**Return type**

None

**addNode()**

**Returns**

the new NodeId

**Return type**

int

**addNodeWithId(id)**

Add a node by choosing a new NodeId.

**Parameters**

**id** (*int*) – The id of the new node

**Raises**

- *pyAgrum.DuplicateElement* (page 287) –
- **If the given id is already used** –

**Return type**

None

**addNodes(n)**

Add n nodes.

**Parameters**

**n** (*int*) – the number of nodes to add.

**Returns**

the new ids

**Return type**

Set of int

**arcs()**

**Returns**

the list of the arcs

**Return type**

List

**children(id)**

**Parameters**

**id** (*int*) – the id of the parent

**Returns**

the set of all the children

**Return type**

Set

**clear()**

Remove all the nodes and arcs from the graph.

**Return type**

None

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**empty()**

Check if the graph is empty.

**Returns**

True if the graph is empty

**Return type**

bool

**emptyArcs()**

Check if the graph doesn't contains arcs.

**Returns**

True if the graph doesn't contains arcs

**Return type**

bool

**eraseArc(*n1*, *n2*)**

Erase the arc between *n1* and *n2*.

**Parameters**

- ***n1*** (*int*) – the id of the tail node
- ***n2*** (*int*) – the id of the head node

**Return type**

None

**eraseChildren(*n*)**

Erase the arcs heading through the node's children.

**Parameters**

***n*** (*int*) – the id of the parent node

**Return type**

None

**eraseNode(*id*)**

Erase the node and all the related arcs.

**Parameters**

***id*** (*int*) – the id of the node

**Return type**

None

**eraseParents(*n*)**

Erase the arcs coming to the node.

**Parameters**

**n** (*int*) – the id of the child node

**Return type**

None

**existsArc(*n1*, *n2*)**

Check if an arc exists bewteen *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Returns**

True if the arc exists

**Return type**

bool

**existsNode(*id*)**

Check if a node with a certain id exists in the graph.

**Parameters**

**id** (*int*) – the checked id

**Returns**

True if the node exists

**Return type**

bool

**hasDirectedPath(*\_from*, *to*)**

Check if a directedpath exists bewteen *from* and *to*.

**Parameters**

- **from** (*int*) – the id of the first node of the (possible) path
- **to** (*int*) – the id of the last node of the (possible) path
- **\_from** (*int*) –

**Returns**

True if the directed path exists

**Return type**

bool

**nodes()****Returns**

the set of ids

**Return type**

set

**parents(*id*)****Parameters**

**id** (*int*) – The id of the child node

**Returns**

the set of the parents ids.

**Return type**

Set

**size()****Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()****Returns**

the number of arcs in the graph

**Return type**

int

**toDot()****Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder(clear=True)****Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**[\*pyAgrum.InvalidDirectedCycle\*](#) (page 288) – If this graph contains cycles**Parameters****clear** (bool) –

## Directed Acyclic Graph

**class pyAgrum.DAG(\*args)**

DAG represents a Directed Graph.

**DAG() -> DAG**

default constructor

**DAG(src) -> DAG****Parameters:**

- **src** (*pyAgrum.DAG*) – the digraph to copy

**addArc(\*args)**

Add an arc from tail to head.

**Parameters**

- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

**Raises**

- [\*pyAgrum.InvalidNode\*](#) (page 289) – If head or tail does not belong to the graph nodes.
- [\*PyAgrum.InvalidDirectedCycle\*](#) – if the arc would create a cycle.

**Return type**

None

**addNode()****Returns**

the new NodeId

**Return type**

int

**addNodeWithId(id)**

Add a node by choosing a new NodeId.

**Parameters****id** (*int*) – The id of the new node**Raises**

- [\*pyAgrum.DuplicateElement\*](#) (page 287) –
- If the given id is already used –

**Return type**

None

**addNodes(*n*)**

Add *n* nodes.

**Parameters**

**n** (*int*) – the number of nodes to add.

**Returns**

the new ids

**Return type**

Set of int

**arcs()****Returns**

the list of the arcs

**Return type**

List

**children(*id*)****Parameters**

**id** (*int*) – the id of the parent

**Returns**

the set of all the children

**Return type**

Set

**clear()**

Remove all the nodes and arcs from the graph.

**Return type**

None

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**dSeparation(\**args*)****Return type**

bool

**empty()**

Check if the graph is empty.

**Returns**

True if the graph is empty

**Return type**

bool

**emptyArcs()**

Check if the graph doesn't contains arcs.

**Returns**

True if the graph doesn't contains arcs

**Return type**

bool

**eraseArc**(*n1*, *n2*)

Erase the arc between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Return type**

None

**eraseChildren**(*n*)

Erase the arcs heading through the node's children.

**Parameters**

- **n** (*int*) – the id of the parent node

**Return type**

None

**eraseNode**(*id*)

Erase the node and all the related arcs.

**Parameters**

- **id** (*int*) – the id of the node

**Return type**

None

**eraseParents**(*n*)

Erase the arcs coming to the node.

**Parameters**

- **n** (*int*) – the id of the child node

**Return type**

None

**existsArc**(*n1*, *n2*)

Check if an arc exists between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Returns**

True if the arc exists

**Return type**

bool

**existsNode**(*id*)

Check if a node with a certain id exists in the graph.

**Parameters**

- **id** (*int*) – the checked id

**Returns**

True if the node exists

**Return type**

bool

**hasDirectedPath**(*\_from*, *to*)

Check if a directed path exists between *from* and *to*.

**Parameters**

- **from** (*int*) – the id of the first node of the (possible) path
- **to** (*int*) – the id of the last node of the (possible) path
- **\_from** (*int*) –

**Returns**

True if the directed path exists

**Return type**

bool

**moralGraph()**

**Return type**

[UndiGraph](#) (page 12)

**moralizedAncestralGraph(nodes)**

**Parameters**

**nodes** (List[int]) –

**Return type**

[UndiGraph](#) (page 12)

**nodes()**

**Returns**

the set of ids

**Return type**

set

**parents(id)**

**Parameters**

**id** (int) – The id of the child node

**Returns**

the set of the parents ids.

**Return type**

Set

**size()**

**Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()**

**Returns**

the number of arcs in the graph

**Return type**

int

**toDot()**

**Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder(clear=True)**

**Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**

[pyAgrum.InvalidDirectedCycle](#) (page 288) – If this graph contains cycles

**Parameters**

**clear** (bool) –

### 1.1.3 Undirected Graphs

#### UndiGraph

**class** pyAgrum.UndiGraph(\*args)

UndiGraph represents an Undirected Graph.

**UndiGraph()** -> UndiGraph

default constructor

**UndiGraph(src)** -> UndiGraph

**Parameters!**

- **src** (*UndiGraph*) – the pyAgrum.UndiGraph to copy

**addEdge(\*args)**

Insert a new edge into the graph.

**Parameters**

- **n1** (*int*) – the id of one node of the new inserted edge
- **n2** (*int*) – the id of the other node of the new inserted edge

**Raises**

[\*pyAgrum.InvalidNode\*](#) (page 289) – If n1 or n2 does not belong to the graph nodes.

**Return type**

None

**addNode()**

**Returns**

the new NodeId

**Return type**

int

**addNodeWithId(id)**

Add a node by choosing a new NodeId.

**Parameters**

**id** (*int*) – The id of the new node

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If the given id is already used

**Return type**

None

**addNodes(n)**

Add n nodes.

**Parameters**

**n** (*int*) – the number of nodes to add.

**Returns**

the new ids

**Return type**

Set of int

**clear()**

Remove all the nodes and edges from the graph.

**Return type**

None

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)



The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**edges()****Returns**

the list of the edges

**Return type**

List

**empty()**

Check if the graph is empty.

**Returns**

True if the graph is empty

**Return type**

bool

**emptyEdges()**

Check if the graph doesn't contains edges.

**Returns**

True if the graph doesn't contains edges

**Return type**

bool

**eraseEdge(*n1*, *n2*)**

Erase the edge between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Return type**

None

**eraseNeighbours(*n*)**

Erase all the edges adjacent to a given node.

**Parameters**

**n** (*int*) – the id of the node

**Return type**

None

**eraseNode(*id*)**

Erase the node and all the adjacent edges.

**Parameters**

**id** (*int*) – the id of the node

**Return type**

None

**existsEdge(*n1*, *n2*)**

Check if an edge exists between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of one extremity of the edge
- **n2** (*int*) – the id of the other extremity if the edge

**Returns**

True if the arc exists

**Return type**

bool

**existsNode(id)**

Check if a node with a certain id exists in the graph.

**Parameters**

**id** (*int*) – the checked id

**Returns**

True if the node exists

**Return type**

bool

**hasUndirectedCycle()**

Checks whether the graph contains cycles.

**Returns**

True if the graph contains a cycle

**Return type**

bool

**neighbours(id)****Parameters**

**id** (*int*) – the id of the checked node

**Returns**

The set of edges adjacent to the given node

**Return type**

Set

**nodes()****Returns**

the set of ids

**Return type**

set

**nodes2ConnectedComponent()****Return type**

Dict[int, int]

**partialUndiGraph(nodes)****Parameters**

- **nodesSet** (*Set*) – The set of nodes composing the partial graph
- **nodes** (*List[int]*) –

**Returns**

The partial graph formed by the nodes given in parameter

**Return type**

[\*pyAgrum.UndiGraph\*](#) (page 12)

**size()****Returns**

the number of nodes in the graph

**Return type**

int

**sizeEdges()****Returns**

the number of edges in the graph

**Return type**

int

**toDot()****Returns**

a friendly display of the graph in DOT format

**Return type**

str

## Clique Graph

**class** pyAgrum.CliqueGraph(\*args)

CliqueGraph represents a Clique Graph.

**CliqueGraph()** -> **CliqueGraph**

default constructor

**CliqueGraph(src)** -> **CliqueGraph**

**Parameter**

- **src** (*pyAgrum.CliqueGraph*) – the CliqueGraph to copy

**addEdge**(*first, second*)

Insert a new edge into the graph.

**Parameters**

- **n1** (*int*) – the id of one node of the new inserted edge
- **n2** (*int*) – the id of the other node of the new inserted edge
- **first** (*int*) –
- **second** (*int*) –

**Raises**

[\*pyAgrum.InvalidNode\*](#) (page 289) – If n1 or n2 does not belong to the graph nodes.

**Return type**

None

**addNode**(*\*args*)

**Returns**

the new NodeId

**Return type**

int

**addNodeWithId**(*id*)

Add a node by choosing a new NodeId.

**Parameters**

**id** (*int*) – The id of the new node

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If the given id is already used

**Return type**

None

**addNodes**(*n*)

Add n nodes.

**Parameters**

**n** (*int*) – the number of nodes to add.

**Returns**

the new ids

**Return type**

Set of int

**addToClique**(*clique\_id, node\_id*)

Change the set of nodes included into a given clique and returns the new set

**Parameters**

- **clique\_id** (*int*) – the id of the clique
- **node\_id** (*int*) – the id of the node

**Raises**

- [\*pyAgrum.NotFound\*](#) (page 289) –
- **If clique\_id does not exist** –
- [\*pyAgrum.DuplicateElement\*](#) (page 287) –
- **If clique\_id set already contains the ndoe** –

**Return type**

None

**clear()**

Remove all the nodes and edges from the graph.

**Return type**

None

**clearEdges()**

Remove all edges and their separators

**Return type**

None

**clique(*clique*)****Parameters**

- **idClique** (*int*) – the id of the clique
- **clique** (*int*) –

**Returns**

The set of nodes included in the clique

**Return type**

Set

**Raises**[\*pyAgrum.NotFound\*](#) (page 289) – If the clique does not belong to the clique graph**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**dict of connected components (as set of nodeIds (*int*)) with a nodeId (root) of each component as key.**Return type**dict(*int*,Set[*int*])**container(*idNode*)****Parameters**

- **idNode** (*int*) – the id of the node

**Returns**

the id of a clique containing the node

**Return type***int***Raises**[\*pyAgrum.NotFound\*](#) (page 289) – If no clique contains *idNode***containerPath(*node1*, *node2*)****Parameters**

- **node1** (*int*) – the id of one node
- **node2** (*int*) – the id of the other node

**Returns**a path from a clique containing *node1* to a clique containing *node2***Return type**

List

**Raises**[\*pyAgrum.NotFound\*](#) (page 289) – If such path cannot be found

**edges()****Returns**

the list of the edges

**Return type**

List

**empty()**

Check if the graph is empty.

**Returns**

True if the graph is empty

**Return type**

bool

**emptyEdges()**

Check if the graph doesn't contains edges.

**Returns**

True if the graph doesn't contains edges

**Return type**

bool

**eraseEdge(*edge*)**

Erase the edge between n1 and n2.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node
- **edge** (*Edge* (page 4)) –

**Return type**

None

**eraseFromClique(*clique\_id*, *node\_id*)**

Remove a node from a clique

**Parameters**

- **clique\_id** (*int*) – the id of the clique
- **node\_id** (*int*) – the id of the node

**Raises***pyAgrum.NotFound* (page 289) – If clique\_id does not exist**Return type**

None

**eraseNeighbours(*n*)**

Erase all the edges adjacent to a given node.

**Parameters****n** (*int*) – the id of the node**Return type**

None

**eraseNode(*node*)**

Erase the node and all the adjacent edges.

**Parameters**

- **id** (*int*) – the id of the node
- **node** (*int*) –

**Return type**

None

**existsEdge(*n1*, *n2*)**

Check if an edge exists between n1 and n2.

**Parameters**

- **n1** (*int*) – the id of one extremity of the edge
- **n2** (*int*) – the id of the other extremity if the edge

**Returns**

True if the arc exists

**Return type**

bool

**existsNode(*id*)**

Check if a node with a certain id exists in the graph.

**Parameters**

**id** (*int*) – the checked id

**Returns**

True if the node exists

**Return type**

bool

**hasRunningIntersection()****Returns**

True if the running intersection property holds

**Return type**

bool

**hasUndirectedCycle()**

Checks whether the graph contains cycles.

**Returns**

True if the graph contains a cycle

**Return type**

bool

**isJoinTree()****Returns**

True if the graph is a join tree

**Return type**

bool

**neighbours(*id*)****Parameters**

**id** (*int*) – the id of the checked node

**Returns**

The set of edges adjacent to the given node

**Return type**

Set

**nodes()****Returns**

the set of ids

**Return type**

set

**nodes2ConnectedComponent()****Return type**

Dict[int, int]

**partialUndiGraph(*nodes*)****Parameters**

- **nodesSet** (*Set*) – The set of nodes composing the partial graph
- **nodes** (*List[int]*) –

**Returns**

The partial graph formed by the nodes given in parameter

**Return type**

[\*pyAgrum.UndiGraph\*](#) (page 12)

**separator**(*cliq1, cliq2*)

**Parameters**

- **edge** ([pyAgrum.Edge](#) (page 4)) – the edge to be checked
- **clique1** (*int*) – one extremity of the edge
- **clique** (*int*) – the other extremity of the edge
- **cliq1** (*int*) –
- **cliq2** (*int*) –

**Returns**

the separator included in a given edge

**Return type**

Set

**Raises**

[pyAgrum.NotFound](#) (page 289) – If the edge does not belong to the clique graph

**setClique**(*idClique, new\_clique*)

changes the set of nodes included into a given clique

**Parameters**

- **idClique** (*int*) – the id of the clique
- **new\_clique** (Set) – the new set of nodes to be included in the clique

**Raises**

[pyAgrum.NotFound](#) (page 289) – If idClique is not a clique of the graph

**Return type**

None

**size**()

**Returns**

the number of nodes in the graph

**Return type**

int

**sizeEdges**()

**Returns**

the number of edges in the graph

**Return type**

int

**toDot**()

**Returns**

a friendly display of the graph in DOT format

**Return type**

str

**toDotWithNames**(*bn*)

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 64)) –
- **network** (a *Bayesian*) –

**Returns**

a friendly display of the graph in DOT format where ids have been changed according to their correspondance in the BN

**Return type**

str

### 1.1.4 Mixed Graph

**class** pyAgrum.MixedGraph(\*args)

MixedGraph represents a graph with both arcs and edges.

**MixedGraph()** -> **MixedGraph**

default constructor

**MixedGraph(src)** -> **MixedGraph**

**Parameters:**

- **src** (*pyAgrum.MixedGraph*) –the MixedGraph to copy

**addArc**(n1, n2)

Add an arc from tail to head.

**Parameters**

- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node
- **n1** (*int*) –
- **n2** (*int*) –

**Raises**

[\*pyAgrum.InvalidNode\*](#) (page 289) – If head or tail does not belong to the graph nodes.

**Return type**

None

**addEdge**(n1, n2)

Insert a new edge into the graph.

**Parameters**

- **n1** (*int*) – the id of one node of the new inserted edge
- **n2** (*int*) – the id of the other node of the new inserted edge

**Raises**

[\*pyAgrum.InvalidNode\*](#) (page 289) – If n1 or n2 does not belong to the graph nodes.

**Return type**

None

**addNode**()

**Returns**

the new NodeId

**Return type**

int

**addNodeWithId**(id)

Add a node by choosing a new NodeId.

**Parameters**

**id** (*int*) – The id of the new node

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If the given id is already used

**Return type**

None

**addNodes**(n)

Add n nodes.

**Parameters**

**n** (*int*) – the number of nodes to add.

**Returns**

the new ids

**Return type**

Set of int



**adjacents(*id*)**

adjacents nodes are neighbours (not oriented), children and parents

**Parameters**

**id** (*int*) – the id of the node

**Returns**

the set of node ids.

**Return type**

set

**arcs()****Returns**

the list of the arcs

**Return type**

List

**children(*id*)****Parameters**

**id** (*int*) – the id of the parent

**Returns**

the set of all the children

**Return type**

Set

**clear()**

Remove all the nodes and edges from the graph.

**Return type**

None

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**edges()****Returns**

the list of the edges

**Return type**

List

**empty()**

Check if the graph is empty.

**Returns**

True if the graph is empty

**Return type**

bool

**emptyArcs()**

Check if the graph doesn't contains arcs.

**Returns**

True if the graph doesn't contains arcs

**Return type**

bool

**emptyEdges()**

Check if the graph doesn't contains edges.

**Returns**

True if the graph doesn't contains edges

**Return type**

bool

**eraseArc(*n1*, *n2*)**

Erase the arc between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Return type**

None

**eraseChildren(*n*)**

Erase the arcs heading through the node's children.

**Parameters**

- **n** (*int*) – the id of the parent node

**Return type**

None

**eraseEdge(*n1*, *n2*)**

Erase the edge between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Return type**

None

**eraseNeighbours(*n*)**

Erase all the edges adjacent to a given node.

**Parameters**

- **n** (*int*) – the id of the node

**Return type**

None

**eraseNode(*id*)**

Erase the node and all the related arcs and edges.

**Parameters**

- **id** (*int*) – the id of the node

**Return type**

None

**eraseParents(*n*)**

Erase the arcs coming to the node.

**Parameters**

- **n** (*int*) – the id of the child node

**Return type**

None

**existsArc(*n1*, *n2*)**

Check if an arc exists between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of the tail node
- **n2** (*int*) – the id of the head node

**Returns**

True if the arc exists

**Return type**

bool

**existsEdge(*n1*, *n2*)**

Check if an edge exists between *n1* and *n2*.

**Parameters**

- **n1** (*int*) – the id of one extremity of the edge
- **n2** (*int*) – the id of the other extremity of the edge

**Returns**

True if the arc exists

**Return type**

bool

**existsNode(*id*)**

Check if a node with a certain id exists in the graph.

**Parameters**

- **id** (*int*) – the checked id

**Returns**

True if the node exists

**Return type**

bool

**hasDirectedPath(*\_from*, *to*)**

Check if a directed path exists between *from* and *to*.

**Parameters**

- **from** (*int*) – the id of the first node of the (possible) path
- **to** (*int*) – the id of the last node of the (possible) path
- **\_from** (*int*) –

**Returns**

True if the directed path exists

**Return type**

bool

**hasUndirectedCycle()**

Checks whether the graph contains cycles.

**Returns**

True if the graph contains a cycle

**Return type**

bool

**mixedOrientedPath(*node1*, *node2*)****Parameters**

- **node1** (*int*) – the id from which the path begins
- **node2** (*int*) – the id to which the path ends

**Returns**

a path from *node1* to *node2*, using edges and/or arcs (following the direction of the arcs). If no path is found, the returned list is empty.

**Return type**

List

**mixedUnorientedPath(*node1*, *node2*)****Parameters**

- **node1** (*int*) – the id from which the path begins
- **node2** (*int*) – the id to which the path ends

**Returns**

a path from *node1* to *node2*, using edges and/or arcs (not necessarily following the direction of the arcs). If no path is found, the list is empty.

**Return type**

List

**neighbours(*id*)****Parameters**

- **id** (*int*) – the id of the checked node

**Returns**

The set of edges adjacent to the given node

**Return type**

Set

**nodes()****Returns**

the set of ids

**Return type**

set

**nodes2ConnectedComponent()****Return type**

Dict[int, int]

**parents(*id*)****Parameters**

**id** (int) – The id of the child node

**Returns**

the set of the parents ids.

**Return type**

Set

**partialUndiGraph(*nodes*)****Parameters**

- **nodesSet** (Set) – The set of nodes composing the partial graph
- **nodes** (List[int]) –

**Returns**

The partial graph formed by the nodes given in parameter

**Return type**

[\*pyAgrum.UndiGraph\*](#) (page 12)

**size()****Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()****Returns**

the number of arcs in the graph

**Return type**

int

**sizeEdges()****Returns**

the number of edges in the graph

**Return type**

int

**toDot()****Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder(*clear=True*)****Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**

[`pyAgrim.InvalidDirectedCycle`](#) (page 288) – If this graph contains cycles

**Parameters**

**clear** (bool) –

## 1.2 Random Variables

aGrUM/pyAgrim is currently dedicated for discrete probability distributions.

There are 5 types of discrete random variables in aGrUM/pyAgrim: `LabelizedVariable`, `DiscretizedVariable`, `IntegerVariable`, `RangeVariable` and `NumericalDiscreteVariable`. The 5 types are mainly provided in order to ease modelization. Derived from `DiscreteVariable`, they share a common API. They essentially differ by the means to create, name and access to their modalities.

### 1.2.1 Common API for Random Discrete Variables

**class** `pyAgrim.DiscreteVariable(*args, **kwargs)`

`DiscreteVariable` is the (abstract) base class for discrete random variables.

**description()****Returns**

the description of the variable

**Return type**

str

**domain()****Returns**

the domain of the variable

**Return type**

str

**domainSize()****Returns**

the number of modalities in the variable domain

**Return type**

int

**empty()****Returns**

True if the domain size < 2

**Return type**

bool

**index(label)****Parameters**

**label** (str) – a label

**Returns**

the indice of the label

**Return type**

int

**label(i)****Parameters**

**i** (int) – the index of the label we wish to return

**Returns**

the indice-th label

**Return type**

str

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label**labels()****Returns**

a tuple containing the labels

**Return type**

tuple

**name()****Returns**

the name of the variable

**Return type**

str

**numerical(*indice*)****Parameters****indice** (*int*) – an index**Returns**

the numerical representation of the indice-th value

**Return type**

float

**setDescription(*theValue*)**

set the description of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**setName(*theValue*)**

sets the name of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**stype()****Returns**

a description of its type

**Return type**

str

**toDiscretizedVar()****Returns**

the discretized variable

**Return type**[\*pyAgrum.DiscretizedVariable\*](#) (page 31)**Raises****pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable**toIntegerVar()****Return type**[\*IntegerVariable\*](#) (page 35)**toLabelizedVar()****Returns**

the labelized variable

**Return type***pyAgrum.LabelizedVariable* (page 27)**Raises****pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable**toNumericalDiscreteVar()****Return type***NumericalDiscreteVariable* (page 42)**toRangeVar()****Returns**

the range variable

**Return type***pyAgrum.RangeVariable* (page 39)**Raises****pyAgrum.RuntimeError** – If the variable is not a RangeVariable**toStringWithDescription()****Returns**

a description of the variable

**Return type**

str

**varType()**

returns the type of variable

**Returns**

the type of the variable.

0: DiscretizedVariable, 1: LabelizedVariable, 2: IntegerVariable, 3: RangeVariable, 4:

**Return type**

int

## 1.2.2 Concrete classes for Random Discrete Variables

### LabelizedVariable

**class** `pyAgrum.LabelizedVariable(*args)`

LabelizedVariable is a discrete random variable with a customizable sequence of labels.

**LabelizedVariable(aName, aDesc="", nbrLabel=2) -> LabelizedVariable****Parameters:**

- **aName** (str) – the name of the variable
- **aDesc** (str) – the (optional) description of the variable
- **nbrLabel** (int) – the number of labels to create (2 by default)

**LabelizedVariable(aName, aDesc="", labels) -> LabelizedVariable****Parameters:**

- **aName** (str) – the name of the variable
- **aDesc** (str) – the (optional) description of the variable
- **labels** (List[str]) – the labels to create

**LabelizedVariable(aLDRV) -> LabelizedVariable****Parameters:**

- **aLDRV** (*pyAgrum.LabelizedVariable*) – The *pyAgrum.LabelizedVariable* that will be copied

## Examples

```
>>> import pyAgrum as gum
>>> # creating a variable with 3 labels : '0', '1' and '2'
>>> va=gum.LabelizedVariable('a','a labeled variable',3)
>>> print(va)
a:Labelized(<0,1,2>)
>>> va.addLabel('foo')
("pyAgrum.LabelizedVariable"@0x7fc4c840dd90) a:Labelized(<0,1,2,foo>)
>>> va.changeLabel(1,'bar')
>>> print(va)
a:Labelized(<0,bar,2,foo>)
>>> vb=gum.LabelizedVariable('b','b',0).addLabel('A').addLabel('B').
↳addLabel('C')
>>> print(vb)
b:Labelized(<A,B,C>)
>>> vb.labels()
('A', 'B', 'C')
>>> vb.isLabel('E')
False
>>> vb.label(2)
'C'
>>> vc=gum.LabelizedVariable('b','b',['one','two','three'])
>>> vc
("pyAgrum.LabelizedVariable"@0x7fc4c840c130) b:Labelized(<one,two,three>
↳)
```

### **addLabel(\*args)**

Add a label with a new index (we assume that we will NEVER remove a label).

#### **Parameters**

**aLabel** (*str*) – the label to be added to the labeled variable

#### **Returns**

the labeled variable

#### **Return type**

*pyAgrum.LabelizedVariable* (page 27)

#### **Raises**

*pyAgrum.DuplicateElement* (page 287) – If the variable already contains the label

### **changeLabel(pos, aLabel)**

Change the label at the specified index

#### **Parameters**

- **pos** (*int*) – the index of the label to be changed
- **aLabel** (*str*) – the label to be added to the labeled variable

#### **Raises**

- *pyAgrum.DuplicateElement* (page 287) – If the variable already contains the new label
- *pyAgrum.OutOfBounds* (page 290) – If the index is greater than the size of the variable

#### **Return type**

None

### **description()**

#### **Returns**

the description of the variable



**Return type**

str

**domain()****Returns**

the domain of the variable as a string

**Return type**

str

**domainSize()****Returns**

the number of modalities in the variable domain

**Return type**

int

**empty()****Returns**

True if the domain size &lt; 2

**Return type**

bool

**eraseLabels()**

Erase all the labels from the variable.

**Return type**

None

**index(*label*)****Parameters****label** (*str*) – a label**Returns**

the indice of the label

**Return type**

int

**isLabel(*aLabel*)**

Indicates whether the variable already has the label passed in argument

**Parameters****aLabel** (*str*) – the label to be tested**Returns**

True if the label already exists

**Return type**

bool

**label(*i*)****Parameters****i** (*int*) – the index of the label we wish to return**Returns**

the indice-th label

**Return type**

str

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label**labels()****Returns**

a tuple containing the labels

**Return type**

tuple

**name()**

**Returns**

the name of the variable

**Return type**

str

**numerical**(*index*)**Parameters**

- **indice** (*int*) – an index
- **index** (*int*) –

**Returns**

the numerical representation of the indice-th value

**Return type**

float

**posLabel**(*label*)**Parameters**

**label** (*str*) –

**Return type**

int

**setDescription**(*theValue*)

set the description of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**setName**(*theValue*)

sets the name of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**stype**()**Returns**

a description of its type

**Return type**

str

**toDiscretizedVar**()**Returns**

the discretized variable

**Return type**

[\*pyAgrum.DiscretizedVariable\*](#) (page 31)

**Raises**

**pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable

**toIntegerVar**()**Return type**

[\*IntegerVariable\*](#) (page 35)

**toLabelizedVar**()**Returns**

the labelized variable

**Return type**

[\*pyAgrum.LabelizedVariable\*](#) (page 27)

**Raises**

**pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable

**toNumericalDiscreteVar()**

**Return type**

*NumericalDiscreteVariable* (page 42)

**toRangeVar()**

**Returns**

the range variable

**Return type**

*pyAgrum.RangeVariable* (page 39)

**Raises**

**pyAgrum.RuntimeError** – If the variable is not a RangeVariable

**toStringWithDescription()**

**Returns**

a description of the variable

**Return type**

str

**varType()**

returns the type of variable

**Returns**

the type of the variable.

0: DiscretizedVariable, 1: LabeledVariable, 2: IntegerVariable, 3: RangeVariable, 4:

**Return type**

int

## DiscretizedVariable

**class pyAgrum.DiscretizedVariable(\*args)**

DiscretizedVariable is a discrete random variable with a set of ticks defining intervals.

**DiscretizedVariable(aName, aDesc, ticks=None, is\_empirical=False) ->**

**DiscretizedVariable`**

**Parameters:**

- **aName** (*str*) – the name of the variable
- **aDesc** (*str*) – the description of the variable
- **\*\*ticks** (*list[float]*) – the list of ticks to add
- **is\_empirical** (*\*bool*) – if False, raise an error if a value is out of bound.

**DiscretizedVariable(aDDRV) -> DiscretizedVariable**

**Parameters:**

- **aDDRV** (*pyAgrum.DiscretizedVariable*) – the *pyAgrum.DiscretizedVariable* that will be copied

## Examples

```
>>> import pyAgrum as gum
>>> vX=gum.DiscretizedVariable('X','X has been discretized').addTick(1).
↳addTick(2).addTick(3).addTick(3.1415)
>>> print(vX)
X:Discretized(<[1;2[, [2;3[, [3;3.1415]>)
>>> vX.isTick(4)
False
>>> vX.labels()
(['1;2[', '2;3[', '3;3.1415'])
>>> # where is the real value 2.5 ?
>>> vX.index('2.5')
1
```

**addTick(\*args)**

**Parameters**

**aTick** (*float*) – the Tick to be added

**Returns**

the discretized variable

**Return type**

*pyAgrum.DiscretizedVariable* (page 31)

**Raises**

*pyAgrum.DefaultInLabel* (page 287) – If the tick is already defined

**description()**

**Returns**

the description of the variable

**Return type**

str

**domain()**

**Returns**

the domain of the variable as a string

**Return type**

str

**domainSize()**

**Returns**

the number of modalities in the variable domain

**Return type**

int

**empty()**

**Returns**

True if the domain size < 2

**Return type**

bool

**eraseTicks()**

erase all the Ticks

**Return type**

None

**index(label)**

**Parameters**

**label** (*str*) – a label

**Returns**

the indice of the label

**Return type**

int

**isEmpirical()****Return type**

bool

**isTick(*aTick*)****Parameters****aTick** (*float*) – the Tick to be tested**Returns**

True if the Tick already exists

**Return type**

bool

**label(*i*)****Parameters****i** (*int*) – the index of the label we wish to return**Returns**

the indice-th label

**Return type**

str

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label**labels()****Returns**

a tuple containing the labels

**Return type**

tuple

**name()****Returns**

the name of the variable

**Return type**

str

**numerical(*indice*)****Parameters****indice** (*int*) – an index**Returns**

the numerical representation of the indice-th value

**Return type**

float

**setDescription(*theValue*)**

set the description of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**setEmpirical(*state*)****Parameters****state** (*bool*) –**Return type**

None

**setName(*theValue*)**

sets the name of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**stype()****Returns**

a description of its type

**Return type**

str

**tick(*i*)**

Indicate the index of the Tick

**Parameters****i** (*int*) – the index of the Tick**Returns****aTick** – the index-th Tick**Return type**

float

**Raises*****pyAgrum.NotFound*** (page 289) – If the index is greater than the number of Ticks**ticks()****Returns**

a tuple containing all the Ticks

**Return type**

tuple

**toDiscretizedVar()****Returns**

the discretized variable

**Return type***pyAgrum.DiscretizedVariable* (page 31)**Raises****pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable**toIntegerVar()****Return type***IntegerVariable* (page 35)**toLabelizedVar()****Returns**

the labelized variable

**Return type***pyAgrum.LabelizedVariable* (page 27)**Raises****pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable**toNumericalDiscreteVar()****Return type***NumericalDiscreteVariable* (page 42)**toRangeVar()****Returns**

the range variable

**Return type***pyAgrum.RangeVariable* (page 39)**Raises****pyAgrum.RuntimeError** – If the variable is not a RangeVariable

**toStringWithDescription()**

**Returns**

a description of the variable

**Return type**

str

**varType()**

returns the type of variable

**Returns**

the type of the variable.

0: DiscretizedVariable, 1: LabelizedVariable, 2: IntegerVariable, 3: RangeVariable, 4:

**Return type**

int

## IntegerVariable

**class** pyAgrum.IntegerVariable(\*args)

IntegerVariable is a discrete random variable with a customizable sequence of int.

**IntegerVariable(aName, aDesc="", values) -> IntegerVariable**

**Parameters:**

- **aName** (str) – the name of the variable
- **aDesc** (str) – the (optional) description of the variable
- **values** (List[int]) – the values to create

**IntegerVariable(aIDRV) -> IntegerVariable**

**Parameters:**

- **aIDRV** (pyAgrum.IntegerVariable) – The pyAgrum.IntegerVariable that will be copied

## Examples

```
>>> import pyAgrum as gum
>>> # creating a variable with 3 values : 1,34,142
>>> va=gum.IntegerVariable('a','a integer variable',[1,34,142])
>>> print(va)
a:Integer(<1,34,142>)
>>> va.addValue(25)
(pyAgrum.IntegerVariable@0000001E4F5D07490) a:Integer(<1,25,34,142>)
>>> va.changeLabel(34,43)
>>> print(va)
a:Integer(<1,25,43,142>)
>>> vb=gum.IntegerVariable('b','b').addValue(34).addValue(142).
↪addValue(1)
>>> print(vb)
b:Integer(<1,34,142>)
>>> vb.labels()
('1', '34', '142')
```

**addValue(\*args)**

Add a value to the list of values for the variable.

**Parameters**

**value** (*int*) – the new value

**Returns**

the Integer variable

**Return type**

*pyAgrum.IntegerVariable* (page 35)

**Raises**

*pyAgrum.DuplicateElement* (page 287) – If the variable already contains the value

**changeValue**(*old\_value*, *new\_value*)

**Parameters**

- **old\_value** (*int*) – the value to be changed
- **new\_value** (*int*) – the new value

**Return type**

None

**description**()

**Returns**

the description of the variable

**Return type**

str

**domain**()

**Returns**

the domain of the variable

**Return type**

str

**domainSize**()

**Returns**

the number of modalities in the variable domain

**Return type**

int

**empty**()

**Returns**

True if the domain size < 2

**Return type**

bool

**eraseValue**(*value*)

**Parameters**

**value** (*int*) – the value to erase. If the value is not in the domain, the function does nothing (no exception raised)

**Return type**

None

**eraseValues**()

Remove all the domain.

**Return type**

None

**index**(*label*)

**Parameters**

**label** (*str*) – a label

**Returns**

the indice of the label

**Return type**

int



**integerDomain()**

**Returns**

the list of integer values that form the domain of this variable

**Return type**

list[int]

**isValue(value)**

**Parameters**

**value** (*int*) – the value to look at.

**Returns**

True if the value is in the domain.

**Return type**

bool

**label(index)**

**Parameters**

- **i** (*int*) – the index of the label we wish to return
- **index** (*int*) –

**Returns**

the indice-th label

**Return type**

str

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label

**labels()**

**Returns**

a tuple containing the labels

**Return type**

tuple

**name()**

**Returns**

the name of the variable

**Return type**

str

**numerical(index)**

**Parameters**

- **indice** (*int*) – an index
- **index** (*int*) –

**Returns**

the numerical representation of the indice-th value

**Return type**

float

**setDescription(theValue)**

set the description of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**setName(theValue)**

sets the name of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**stype()**

**Returns**  
a description of its type  
**Return type**  
str

**toDiscretizedVar()**

**Returns**  
the discretized variable  
**Return type**  
[\*pyAgrum.DiscretizedVariable\*](#) (page 31)  
**Raises**  
**pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable

**toIntegerVar()**

**Return type**  
[\*IntegerVariable\*](#) (page 35)

**toLabelizedVar()**

**Returns**  
the labeled variable  
**Return type**  
[\*pyAgrum.LabelizedVariable\*](#) (page 27)  
**Raises**  
**pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable

**toNumericalDiscreteVar()**

**Return type**  
[\*NumericalDiscreteVariable\*](#) (page 42)

**toRangeVar()**

**Returns**  
the range variable  
**Return type**  
[\*pyAgrum.RangeVariable\*](#) (page 39)  
**Raises**  
**pyAgrum.RuntimeError** – If the variable is not a RangeVariable

**toStringWithDescription()**

**Returns**  
a description of the variable  
**Return type**  
str

**varType()**

returns the type of variable  
**Returns**  
the type of the variable.  
0: DiscretizedVariable, 1: LabelizedVariable, 2: IntegerVariable, 3: RangeVariable, 4:  
**Return type**  
int

## RangeVariable

**class** pyAgrum.**RangeVariable**(\*args)

RangeVariable represents a variable with a range of integers as domain.

**RangeVariable(aName, aDesc, minVal, maxVal) -> RangeVariable**

**Parameters:**

- **aName** (*str*) – the name of the variable
- **aDesc** (*str*) – the description of the variable
- **minVal** (*int*) – the minimal integer of the interval
- **maxVal** (*int*) – the maximal integer of the interval

**RangeVariable(aName, aDesc='') -> RangeVariable**

**Parameters:**

- **aName** (*str*) – the name of the variable
- **aDesc** (*str*) – the description of the variable

By default minVal=0 and maxVal=1

**RangeVariable(aRV) -> RangeVariable**

**Parameters:**

- **aDV** (*RangeVariable*) – the pyAgrum.RangeVariable that will be copied

## Examples

```
>>> import pyAgrum as gum
>>> vI=gum.RangeVariable('I','I in [4,10]',4,10)
>>> print(vI)
I:Range([4,10])
>>> vI.maxVal()
10
>>> vI.belongs(1)
False
>>> # where is the value 5 ?
>>> vI.index('5')
1
>>> vI.labels()
('4', '5', '6', '7', '8', '9', '10')
```

**belongs(val)**

**Parameters**

**val** (*int*) – the value to be tested

**Returns**

True if the value in parameters belongs to the variable's interval.

**Return type**

bool

**description()**

**Returns**

the description of the variable

**Return type**

str

**domain()**

**Returns**

the domain of the variable

**Return type**

str

**domainSize()**

**Returns**

the number of modalities in the variable domain

**Return type**

int

**empty()**

**Returns**

True if the domain size < 2

**Return type**

bool

**index(*arg2*)**

**Parameters**

**arg2** (*str*) – a label

**Returns**

the indice of the label

**Return type**

int

**label(*index*)**

**Parameters**

- **indice** (*int*) – the index of the label we wish to return

- **index** (*int*) –

**Returns**

the indice-th label

**Return type**

str

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label

**labels()**

**Returns**

a tuple containing the labels

**Return type**

tuple

**maxVal()**

**Returns**

the upper bound of the variable.

**Return type**

int

**minVal()**

**Returns**

the lower bound of the variable

**Return type**

int

**name()**

**Returns**

the name of the variable

**Return type**

str

**numerical**(*index*)

**Parameters**

- **indice** (*int*) – an index
- **index** (*int*) –

**Returns**

the numerical representation of the indice-th value

**Return type**

float

**setDescription**(*theValue*)

set the description of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**setMaxVal**(*maxVal*)

Set a new value of the upper bound

**Parameters**

**maxVal** (*int*) – The new value of the upper bound

**Warning:** An error should be raised if the value is lower than the lower bound.

**Return type**

None

**setMinVal**(*minVal*)

Set a new value of the lower bound

**Parameters**

**minVal** (*int*) – The new value of the lower bound

**Warning:** An error should be raised if the value is higher than the upper bound.

**Return type**

None

**setName**(*theValue*)

sets the name of the variable.

**Parameters**

**theValue** (*str*) – the new description of the variable

**Return type**

None

**stype**()

**Returns**

a description of its type

**Return type**

str

**toDiscretizedVar**()

**Returns**

the discretized variable

**Return type**

[\*pyAgrum.DiscretizedVariable\*](#) (page 31)

**Raises**

**pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable

**toIntegerVar**()

**Return type***IntegerVariable* (page 35)**toLabelizedVar()****Returns**

the labelized variable

**Return type***pyAgrum.LabelizedVariable* (page 27)**Raises****pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable**toNumericalDiscreteVar()****Return type***NumericalDiscreteVariable* (page 42)**toRangeVar()****Returns**

the range variable

**Return type***pyAgrum.RangeVariable* (page 39)**Raises****pyAgrum.RuntimeError** – If the variable is not a RangeVariable**toStringWithDescription()****Returns**

a description of the variable

**Return type**

str

**varType()**

returns the type of variable

**Returns**

the type of the variable.

0: DiscretizedVariable, 1: LabelizedVariable, 2: IntegerVariable, 3: RangeVariable, 4:

**Return type**

int

**NumericalDiscreteVariable****class** `pyAgrum.NumericalDiscreteVariable(*args)`**addValue(\*args)**

Add a value to the list of values for the variable.

**Parameters****value** (*float*) – the new value**Returns**

the Integer variable

**Return type***pyAgrum.IntegerVariable* (page 35)**Raises****pyAgrum.DuplicateElement** (page 287) – If the variable already contains the value**changeValue(old\_value, new\_value)****Parameters**

- **old\_value** (*float*) –
- **new\_value** (*float*) –

**Return type**

None

**closestIndex**(*val*)**Parameters****val** (float) –**Return type**

int

**closestLabel**(*val*)**Parameters****val** (float) –**Return type**

str

**description**()**Returns**

the description of the variable

**Return type**

str

**domain**()**Returns**

the domain of the variable

**Return type**

str

**domainSize**()**Returns**

the number of modalities in the variable domain

**Return type**

int

**empty**()**Returns**

True if the domain size &lt; 2

**Return type**

bool

**eraseValue**(*value*)**Parameters****value** (float) –**Return type**

None

**eraseValues**()**Return type**

None

**index**(*label*)**Parameters****label** (*str*) – a label**Returns**

the indice of the label

**Return type**

int

**isValue**(*value*)**Parameters****value** (float) –

**Return type**

bool

**label**(*index*)**Parameters**

- **i** (*int*) – the index of the label we wish to return
- **index** (*int*) –

**Returns**

the indice-th label

**Return type**

str

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If the variable does not contain the label**labels**()**Returns**

a tuple containing the labels

**Return type**

tuple

**name**()**Returns**

the name of the variable

**Return type**

str

**numerical**(*index*)**Parameters**

- **indice** (*int*) – an index
- **index** (*int*) –

**Returns**

the numerical representation of the indice-th value

**Return type**

float

**numericalDomain**()**Return type**

object

**setDescription**(*theValue*)

set the description of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**setName**(*theValue*)

sets the name of the variable.

**Parameters****theValue** (*str*) – the new description of the variable**Return type**

None

**stype**()**Returns**

a description of its type

**Return type**

str

**toDiscretizedVar**()



**Returns**

the discretized variable

**Return type***pyAgrum.DiscretizedVariable* (page 31)**Raises****pyAgrum.RuntimeError** – If the variable is not a DiscretizedVariable**toIntegerVar()****Return type***IntegerVariable* (page 35)**toLabelizedVar()****Returns**

the labeled variable

**Return type***pyAgrum.LabelizedVariable* (page 27)**Raises****pyAgrum.RuntimeError** – If the variable is not a LabelizedVariable**toNumericalDiscreteVar()****Return type***NumericalDiscreteVariable* (page 42)**toRangeVar()****Returns**

the range variable

**Return type***pyAgrum.RangeVariable* (page 39)**Raises****pyAgrum.RuntimeError** – If the variable is not a RangeVariable**toStringWithDescription()****Returns**

a description of the variable

**Return type**

str

**varType()**

returns the type of variable

**Returns**

the type of the variable.

0: DiscretizedVariable, 1: LabelizedVariable, 2: IntegerVariable, 3: RangeVariable, 4:

**Return type**

int

## 1.3 Potential and Instantiation

*pyAgrum.Potential* (page 53) is a multi-dimensional array with a *pyAgrum.DiscreteVariable* (page 25) associated to each dimension. It is used to represent probabilities and utilities tables in aGrUMs' multidimensional (graphical) models with some conventions.

- The data are stored by iterating over each variable in the sequence.

```
>>> a=gum.RangeVariable("A","variable A",1,3)
>>> b=gum.RangeVariable("B","variable B",1,2)
>>> p=gum.Potential().add(a).add(b).fillWith([1,2,3,4,5,6])
```

(continues on next page)

(continued from previous page)

```
>>> print(p)
      || A
B      || 1      | 2      | 3      |
-----||-----|-----|-----|
1      || 1.0000 | 2.0000 | 3.0000 |
2      || 4.0000 | 5.0000 | 6.0000 |
```

- If a `pyAgrum.Potential` (page 53) with the sequence of `pyAgrum.DiscreteVariable` (page 25) X,Y,Z represents a conditional probability Table (CPT), it will be  $P(X|Y,Z)$ .

```
>>> print(p.normalizeAsCPT())
      || A
B      || 1      | 2      | 3      |
-----||-----|-----|-----|
1      || 0.1667 | 0.3333 | 0.5000 |
2      || 0.2667 | 0.3333 | 0.4000 |
```

- For addressing and looping in a `pyAgrum.Potential` (page 53) structure, pyAgrum provides `Instantiation` class which represents a multi-dimensionnal index.

```
>>> I=gum.Instantiation(p)
>>> print(I)
<A:1|B:1>
>>> I.inc();print(I)
<A:2|B:1>
>>> I.inc();print(I)
<A:3|B:1>
>>> I.inc();print(I)
<A:1|B:2>
>>> I.setFirst();print(f"{I} -> {p.get(I)}")
<A:1|B:1> -> 0.16666666666666666
>>> I["B"]="2";print(f"{I} -> {p.get(I)}")
<A:1|B:2> -> 0.26666666666666666
```

- `pyAgrum.Potential` (page 53) include tensor operators (see for instance this [notebook](http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/potentials.ipynb.html) (<http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/potentials.ipynb.html>)).

```
>>> c=gum.RangeVariable("C","variable C",1,5)
>>> q=gum.Potential().add(a).add(c).fillWith(1)
>>> print(p+q)
      || A
C      || B      || 1      | 2      | 3      |
-----||-----|-----|-----|
1      || 1      || 1.1667 | 1.3333 | 1.5000 |
2      || 1      || 1.1667 | 1.3333 | 1.5000 |
3      || 1      || 1.1667 | 1.3333 | 1.5000 |
4      || 1      || 1.1667 | 1.3333 | 1.5000 |
5      || 1      || 1.1667 | 1.3333 | 1.5000 |
1      || 2      || 1.2667 | 1.3333 | 1.4000 |
2      || 2      || 1.2667 | 1.3333 | 1.4000 |
3      || 2      || 1.2667 | 1.3333 | 1.4000 |
4      || 2      || 1.2667 | 1.3333 | 1.4000 |
5      || 2      || 1.2667 | 1.3333 | 1.4000 |
>>> print((p*q).margSumOut(["B","C"])) # marginalize p*q over B and C(using
→ sum)
      A
1      | 2      | 3      |
```

(continues on next page)

(continued from previous page)

-----   -----   -----
2.1667   3.3333   4.5000

### 1.3.1 Instantiation

**class** pyAgrum.**Instantiation**(\*args)

Class for assigning/browsing values to tuples of discrete variables.

Instantiation is designed to assign values to tuples of variables and to efficiently loop over values of subsets of variables.

**Instantiation()** -> **Instantiation**

default constructor

**Instantiation(aI)** -> **Instantiation**

**Parameters:**

- **aI** (*pyAgrum.Instantiation*) – the Instantiation we copy

**Returns**

- *pyAgrum.Instantiation* – An empty tuple or a copy of the one in parameters
- *Instantiation is subscriptable therefore values can be easily accessed/modified.*

#### Examples

```
>>> ## Access the value of A in an instantiation aI
>>> valueOfA = aI['A']
>>> ## Modify the value
>>> aI['A'] = newValueOfA
```

**add(v)**

Adds a new variable in the Instantiation.

**Parameters**

**v** (*pyAgrum.DiscreteVariable* (page 25)) – The new variable added to the Instantiation

**Raises**

*DuplicateElement* (page 287) – If the variable is already in this Instantiation

**Return type**

None

**addVarsFromModel(model, names)**

From a graphical model, add all the variable whose names are in the iterable

**Parameters**

- **model** (*pyAgrum.GraphicalModel*) –
- **network** (*Markov*) –
- **network** –
- **Diagram** (*Influence*) –
- **etc.** –
- **names** (*iterable of strings*) –
- **string** (*a list/set/etc of names of variables (as)*) –

**Returns**

- *pyAgrum.Instantiation*
- *the current instantiation (self) in order to chain methods.*

**chgVal(\*args)**

Assign newval to v (or to the variable at position varPos) in the Instantiation.

**Parameters**

- **v** ([pyAgrum.DiscreteVariable](#) (page 25) or *string*) – The variable whose value is assigned (or its name)
- **varPos** (*int*) – The index of the variable whose value is assigned in the tuple of variables of the Instantiation
- **newval** (*int* or *string*) – The index of the value assigned (or its name)

**Returns**

The modified instantiation

**Return type**

[pyAgrum.Instantiation](#) (page 47)

**Raises**

- **NotFound** (page 289) – If variable v does not belong to the instantiation.
- **OutOfBounds** (page 290) – If newval is not a possible value for the variable.

**clear()**

Erase all variables from an Instantiation.

**Return type**

None

**contains(\*args)**

Indicates whether a given variable belongs to the Instantiation.

**Parameters**

- **v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable for which the test is made.

**Returns**

True if the variable is in the Instantiation.

**Return type**

bool

**dec()**

Operator –.

**Return type**

None

**decIn(i)**

Operator – for the variables in i.

**Parameters**

- **i** ([pyAgrum.Instantiation](#) (page 47)) – The set of variables to decrement in this Instantiation

**Return type**

None

**decNotVar(v)**

Operator – for vars which are not v.

**Parameters**

- **v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable not to decrement in this Instantiation.

**Return type**

None

**decOut(i)**

Operator – for the variables not in i.

**Parameters**

- **i** ([pyAgrum.Instantiation](#) (page 47)) – The set of variables to not decrement in this Instantiation.

**Return type**

None

**decVar(*v*)**

Operator – for variable *v* only.

**Parameters**

**v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable to decrement in this Instantiation.

**Raises**

[NotFound](#) (page 289) – If variable *v* does not belong to the Instantiation.

**Return type**

None

**domainSize()****Returns**

The product of the variable's domain size in the Instantiation.

**Return type**

int

**empty()****Returns**

True if the instantiation is empty.

**Return type**

bool

**end()****Returns**

True if the Instantiation reached the end.

**Return type**

bool

**erase(\**args*)****Parameters**

**v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable to be removed from this Instantiation.

**Raises**

[NotFound](#) (page 289) – If *v* does not belong to this Instantiation.

**Return type**

None

**fromdict(*dict*)**

Change the values in an instantiation from a dictionary *{variable\_name:value}* where value can be a position (int) or a label (string).

If a variable\_name does not occur in the instantiation, nothing is done.

**Warning:** OutOfBounds raised if a value cannot be found.

**Parameters**

**dict** (object) –

**Return type**

None

**hamming()****Returns**

the hamming distance of this instantiation.

**Return type**

int

**inOverflow()****Returns**

True if the current value of the tuple is correct

**Return type**

bool

**inc()**

Operator ++.

**Return type**

None

**incIn(*i*)**Operator ++ for the variables in *i*.**Parameters****i** ([pyAgrum.Instantiation](#) (page 47)) – The set of variables to increment in this Instantiation.**Return type**

None

**incNotVar(*v*)**Operator ++ for vars which are not *v*.**Parameters****v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable not to increment in this Instantiation.**Return type**

None

**incOut(*i*)**Operator ++ for the variables not in *i*.**Parameters****i** ([Instantiation](#) (page 47)) – The set of variable to not increment in this Instantiation.**Return type**

None

**incVar(*v*)**Operator ++ for variable *v* only.**Parameters****v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable to increment in this Instantiation.**Raises**[NotFound](#) (page 289) – If variable *v* does not belong to the Instantiation.**Return type**

None

**isMutable()****Return type**

bool

**nbrDim()****Returns**

The number of variables in the Instantiation.

**Return type**

int

**pos(*v*)****Returns**the position of the variable *v*.**Return type**

int

**Parameters****v** ([pyAgrum.DiscreteVariable](#) (page 25)) – the variable for which its position is return.

**Raises**

**NotFound** (page 289) – If *v* does not belong to the instantiation.

**rend()****Returns**

True if the Instantiation reached the rend.

**Return type**

bool

**reorder(\*args)**

Reorder vars of this instantiation giving the order in *v* (or *i*).

**Parameters**

- **i** ([pyAgrum.Instantiation](#) (page 47)) – The sequence of variables with which to reorder this Instantiation.
- **v** (*list*) – The new order of variables for this Instantiation.

**Return type**

None

**setFirst()**

Assign the first values to the tuple of the Instantiation.

**Return type**

None

**setFirstIn(i)**

Assign the first values in the Instantiation for the variables in *i*.

**Parameters**

- **i** ([pyAgrum.Instantiation](#) (page 47)) – The variables to which their first value is assigned in this Instantiation.

**Return type**

None

**setFirstNotVar(v)**

Assign the first values to variables different of *v*.

**Parameters**

- **v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable that will not be set to its first value in this Instantiation.

**Return type**

None

**setFirstOut(i)**

Assign the first values in the Instantiation for the variables not in *i*.

**Parameters**

- **i** ([pyAgrum.Instantiation](#) (page 47)) – The variable that will not be set to their first value in this Instantiation.

**Return type**

None

**setFirstVar(v)**

Assign the first value in the Instantiation for var *v*.

**Parameters**

- **v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable that will be set to its first value in this Instantiation.

**Return type**

None

**setLast()**

Assign the last values in the Instantiation.

**Return type**

None

**setLastIn(*i*)**

Assign the last values in the Instantiation for the variables in *i*.

**Parameters**

**i** ([pyAgrum.Instantiation](#) (page 47)) – The variables to which their last value is assigned in this Instantiation.

**Return type**

None

**setLastNotVar(*v*)**

Assign the last values to variables different of *v*.

**Parameters**

**v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable that will not be set to its last value in this Instantiation.

**Return type**

None

**setLastOut(*i*)**

Assign the last values in the Instantiation for the variables not in *i*.

**Parameters**

**i** ([pyAgrum.Instantiation](#) (page 47)) – The variables that will not be set to their last value in this Instantiation.

**Return type**

None

**setLastVar(*v*)**

Assign the last value in the Instantiation for var *v*.

**Parameters**

**v** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable that will be set to its last value in this Instantiation.

**Return type**

None

**setMutable()****Return type**

None

**setVals(*i*)**

Assign the values from *i* in the Instantiation.

**Parameters**

**i** ([pyAgrum.Instantiation](#) (page 47)) – An Instantiation in which the new values are searched

**Returns**

a reference to the instantiation

**Return type**

[pyAgrum.Instantiation](#) (page 47)

**todict(*withLabels=True*)**

Create a dictionary *{variable\_name:value}* from an instantiation

**Parameters**

**withLabels** (*boolean*) – The value will be a label (string) if True. It will be a position (int) if False.

**Returns**

The dictionary

**Return type**

Dict[str,int]

**unsetEnd()**

Alias for unsetOverflow().

**Return type**

None



**unsetOverflow()**

Removes the flag overflow.

**Return type**

None

**val(\*args)****Parameters**

- **i** (*int*) – The index of the variable.
- **var** ([pyAgrim.DiscreteVariable](#) (page 25)) – The variable the value of which we wish to know

**Returns**

the current value of the variable.

**Return type**

int

**Raises**

[NotFound](#) (page 289) – If the element cannot be found.

**variable(\*args)****Parameters**

**i** (*int*) – The index of the variable

**Returns**

the variable at position i in the tuple.

**Return type**

[pyAgrim.DiscreteVariable](#) (page 25)

**Raises**

[NotFound](#) (page 289) – If the element cannot be found.

**variablesSequence()****Returns**

a list containing the sequence of variables

**Return type**

list

### 1.3.2 Potential

**class pyAgrim.Potential(\*args)**

Class representing a potential.

**Potential() -> Potential**

default constructor

**Potential(src) -> Potential****Parameters:**

- **src** ([pyAgrim.Potential](#)) – the Potential to copy

**KL(p)**

Check the compatibility and compute the Kullback-Leibler divergence between the potential and.

**Parameters**

**p** ([pyAgrim.Potential](#) (page 53)) – the potential from which we want to calculate the divergence.

**Returns**

The value of the divergence

**Return type**

float

**Raises**

- [pyAgrim.InvalidArgument](#) (page 288) – If p is not compatible with the potential (dimension, variables)

- [\*pyAgrum.FatalError\*](#) (page 287) – If a zero is found in *p* or the potential and not in the other.

**abs()**

Apply abs on every element of the container

**Returns**

a reference to the modified potential.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**add(*v*)**

Add a discrete variable to the potential.

**Parameters**

**v** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the var to be added

**Raises**

- [\*DuplicateElement\*](#) (page 287) – If the variable is already in this Potential.
- [\*InvalidArgument\*](#) (page 288) – If the variable is empty.

**Returns**

a reference to the modified potential.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**argmax()****Returns**

the list of positions of the max and the max of all elements in the Potential

**Return type**

Tuple[Dict[str,int],float]

**argmin()****Returns**

the list of positions of the min and the min of all elements in the Potential

**Return type**

Tuple[Dict[str,int],float]

**contains(*v*)****Parameters**

**v** ([\*pyAgrum.Potential\*](#) (page 53)) – a DiscreteVariable.

**Returns**

True if the var is in the potential

**Return type**

bool

**domainSize()****Return type**

int

**draw()**

draw a value using the potential as a probability table.

**Returns**

the index of the drawn value

**Return type**

int

**empty()****Returns**

Returns true if no variable is in the potential.

**Return type**

bool

**entropy()**

**Returns**

the entropy of the potential

**Return type**

float

**extract(\*args)**

create a new Potential extracted from self given a partial instantiation.

**Parameters**

- **inst** (*pyAgrum.instantiation*) – a partial instantiation
- **dict** (*Dict[str, str|int]*) – a dictionary containing values for some discrete variables.

**Warning:** if the dictionary contains a key that is not the name of a variable in the *pyAgrum.Potential*, this key is just not used without notification. Then *pyAgrum.Potential.extract* concerns only the variables that both are in the Potential and in the dictionary.

**Returns**

the new Potential

**Return type**

*pyAgrum.Potential* (page 53)

**fillWith(\*args)**

Automatically fills the potential with v.

**Parameters**

- **v** (*number or list of values or pyAgrum.Potential* (page 53)) – a value or a list/pyAgrum.Potential containing the values to fill the Potential with.
- **mapping** (*list/tuple/dict*) –

**Warning:**

- if v is a list, the size of the list must be the size of the potential
- if v is a ref:pyAgrum.Potential, it must contain variables with exactly the same names and labels but not necessarily the same variables. If
- If the second argument *mapping* is given, *mapping* explains how to map the variables of the potential source to the variables of the potential destination.
- If *mapping* is a sequence, the order follows the same order as *destination.names*. If *mapping* is a dict, the keys are the names in the destination and the values are the names in the source.

**Returns**

a reference to the modified potential

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

- **pyAgrum.SizeError** (page 290) – If v size's does not matches the domain size.
- **pyAgrum.ArgumentError** (page 290) – If anything wrong with the arguments.

**fillWithFunction(s, noise=None)**

Automatically fills the potential as a (quasi) deterministic CPT with the evaluation of the expression s.

The expression s gives a value for the first variable using the names of the last variables. The computed CPT is deterministic unless noise is used to add a 'probabilistic' noise around the exact value given by the expression.

## Examples

```
>>> import pyAgrum as gum
>>> bn=gum.fastBN("A[3]->B[3]<-C[3]")
>>> bn.cpt("B").fillWithFunction("(A+C)/2")
```

### Parameters

- **s** (*str*) – an expression using the name of the last variables of the Potential and giving a value to the first variable of the Potential
- **noise** (*list*) – an (odd) list of numerics giving a pattern of ‘probabilistic noise’ around the value.

**Warning:** The expression may have any numerical values, but will be then transformed to the closest correct value for the range of the variable.

### Returns

a reference to the modified potential

### Return type

*pyAgrum.Potential* (page 53)

### Raises

- *pyAgrum.InvalidArgument* (page 288) –
- If the first variable is Labelized or Integer, or if the len of the noise is not odd. –

## findAll(*v*)

### Parameters

**v** (*float*) –

### Return type

List[Dict[str, int]]

## get(*i*)

### Parameters

**i** (*pyAgrum.Instantiation* (page 47)) – an Instantiation

### Returns

the value in the Potential at the position given by the instantiation

### Return type

float

## inverse()

### Return type

*Potential* (page 53)

## isNonZeroMap()

### Returns

a boolean-like potential using the predicate *isNonZero*.

### Return type

*pyAgrum.Potential* (page 53)

## log2()

log2 all the values in the Potential

**Warning:** When the Potential contains 0 or negative values, no exception are raised but *-inf* or *nan* values are assigned.

### Return type

*Potential* (page 53)

**loopIn()**

Generator to iterate inside a Potential.

Yield an pyAgrum.Instantiation that iterates over all the possible values for the pyAgrum.Potential

**Examples**

```
>>> import pyAgrum as gum
>>> bn=gum.fastBN("A[3]->B[3]<-C[3]")
>>> for i in bn.cpt("B").loopIn():
    print(i)
    print(bn.cpt("B").get(i))
    bn.cpt("B").set(i,0.3)
```

**margMaxIn(varnames)**

Projection using *max* as operation.

**Parameters**

**varnames** (*set*) – the set of vars to keep

**Returns**

the projected Potential

**Return type**

*pyAgrum.Potential* (page 53)

**margMaxOut(varnames)**

Projection using *max* as operation.

**Parameters**

**varnames** (*set*) – the set of vars to eliminate

**Returns**

the projected Potential

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

*pyAgrum.InvalidArgument* (page 288) – If varnames contains only one variable that does not exist in the Potential

**margMinIn(varnames)**

Projection using *min* as operation.

**Parameters**

**varnames** (*set*) – the set of vars to keep

**Returns**

the projected Potential

**Return type**

*pyAgrum.Potential* (page 53)

**margMinOut(varnames)**

Projection using *min* as operation.

**Parameters**

**varnames** (*set*) – the set of vars to eliminate

**Returns**

the projected Potential

**Return type**

*pyAgrum.Potential* (page 53)

**Warning:** InvalidArgument raised if varnames contains only one variable that does not exist in the Potential

**margProdIn**(*varnames*)

Projection using multiplication as operation.

**Parameters**

**varnames** (*set*) – the set of vars to keep

**Returns**

the projected Potential

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**margProdOut**(*varnames*)

Projection using multiplication as operation.

**Parameters**

**varnames** (*set*) – the set of vars to eliminate

**Returns**

the projected Potential

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If varnames contains only one variable that does not exist in the Potential

**margSumIn**(*varnames*)

Projection using sum as operation.

**Parameters**

**varnames** (*set*) – the set of vars to keep

**Returns**

the projected Potential

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**margSumOut**(*varnames*)

Projection using sum as operation.

**Parameters**

**varnames** (*set*) – the set of vars to eliminate

**Returns**

the projected Potential

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If varnames contains only one variable that does not exist in the Potential

**max**()**Returns**

the maximum of all elements in the Potential

**Return type**

float

**maxNonOne**()**Returns**

the maximum of non one elements in the Potential

**Return type**

float

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – If all value == 1.0

**min**()**Returns**

the min of all elements in the Potential

**Return type**

float

**minNonZero()****Returns**

the min of non zero elements in the Potential

**Return type**

float

**Raises**[\*pyAgrum.NotFound\*](#) (page 289) – If all value == 0.0**property names****Returns**

a list containing the name of each variables in the potential

**Return type**

list

**Warning:** listed in the reverse order of the enumeration order of the variables.**nbrDim(\*args)****Returns**

the number of vars in the multidimensional container.

**Return type**

int

**newFactory()**

Erase the Potential content and create a new empty one.

**Returns**

a reference to the new Potential

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**new\_abs()****Return type**[\*Potential\*](#) (page 53)**new\_log2()****Return type**[\*Potential\*](#) (page 53)**new\_sq()****Return type**[\*Potential\*](#) (page 53)**noising(alpha)****Parameters****alpha** (float) –**Return type**[\*Potential\*](#) (page 53)**normalize()**

Normalize the Potential (do nothing if sum is 0)

**Returns**

a reference to the normalized Potential

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**normalizeAsCPT(varId=0)**

Normalize the Potential as a CPT

**Returns**

a reference to the normalized Potential

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

*pyAgrum.FatalError* (page 287) – If some distribution sums to 0

**Parameters**

**varId** (int) –

**pos(v)****Parameters**

**v** (*pyAgrum.DiscreteVariable* (page 25)) – The variable for which the index is returned.

**Return type**

Returns the index of a variable.

**Raises**

*pyAgrum.NotFound* (page 289) – If v is not in this multidimensional matrix.

**product()****Returns**

the product of all elements in the Potential

**Return type**

float

**putFirst(varname)****Parameters**

- **v** (*pyAgrum.DiscreteVariable* (page 25)) – The variable for which the index should be 0.
- **varname** (str) –

**Returns**

a reference to the modified potential

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

*pyAgrum.InvalidArgument* (page 288) – If the var is not in the potential

**random()****Return type**

*Potential* (page 53)

**randomCPT()****Return type**

*Potential* (page 53)

**randomDistribution()****Return type**

*Potential* (page 53)

**remove(var)****Parameters**

**v** (*pyAgrum.DiscreteVariable* (page 25)) – The variable to be removed

**Returns**

a reference to the modified potential

**Return type**

*pyAgrum.Potential* (page 53)

<b>Warning:</b> IndexError raised if the var is not in the potential
--

**Parameters**

**var** (*DiscreteVariable* (page 25)) –



**reorganize(\*args)**

Create a new Potential with another order.

**Returns**

**varnames** – a list of the var names in the new order

**Return type**

list

**Returns**

a reference to the modified potential

**Return type**

*pyAgrum.Potential* (page 53)

**scale(v)**

Create a new potential multiplied by v.

**Parameters**

**v** (*float*) – a multiplier

**Return type**

a reference to the modified potential

**set(i, value)**

Change the value pointed by i

**Parameters**

- **i** (*pyAgrum.Instantiation* (page 47)) – The Instantiation to be changed
- **value** (*float*) – The new value of the Instantiation

**Return type**

None

**property shape****Returns**

a list containing the dimensions of each variables in the potential

**Return type**

list

**Warning:** *p.shape* and *p[:,].shape* list the dimensions in different order

**sq()**

Square all the values in the Potential

**Return type**

*Potential* (page 53)

**sum()****Returns**

the sum of all elements in the Potential

**Return type**

float

**property thisown**

The membership flag

**toarray()****Returns**

the potential as an array

**Return type**

array

**toclipboard(\*\*kwargs)**

Write a text representation of object to the system clipboard. This can be pasted into spreadsheet, for instance.

**tolatex()**

Render object to a LaTeX tabular.

Requires to include *booktabs* package in the LaTeX document.

**Returns**

the potential as LaTeX string

**Return type**

str

**tolist()****Returns**

the potential as a list

**Return type**

list

**topandas()****Returns**

the potential as an pandas.DataFrame

**Return type**

pandas.DataFrame

**translate(v)**

Create a new potential added with v.

**Parameters**

**v** (*float*) – The value to be added

**Return type**

a reference to the modified potential

**property var\_dims****Returns**

a list containing the dimensions of each variables in the potential

**Return type**

list

**Warning:** This methods is deprecated. Please use `gum.Potential.shape` and note the change in the order !

`var_dims` return a list in the reverse order of the enumeration order of the variables.

**property var\_names****Returns**

a list containing the name of each variables in the potential

**Return type**

list

**Warning:** This methods is deprecated. Please use `gum.Potential.names` and note the change in the order !

`var_names` return a list in the reverse order of the enumeration order of the variables.

**variable(\*args)****Parameters**

**i** (*int*) – An index of this multidimensional matrix.

**Return type**

the variable at the ith index

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – If i does not reference a variable in this multidimensional matrix.

**variablesSequence()**

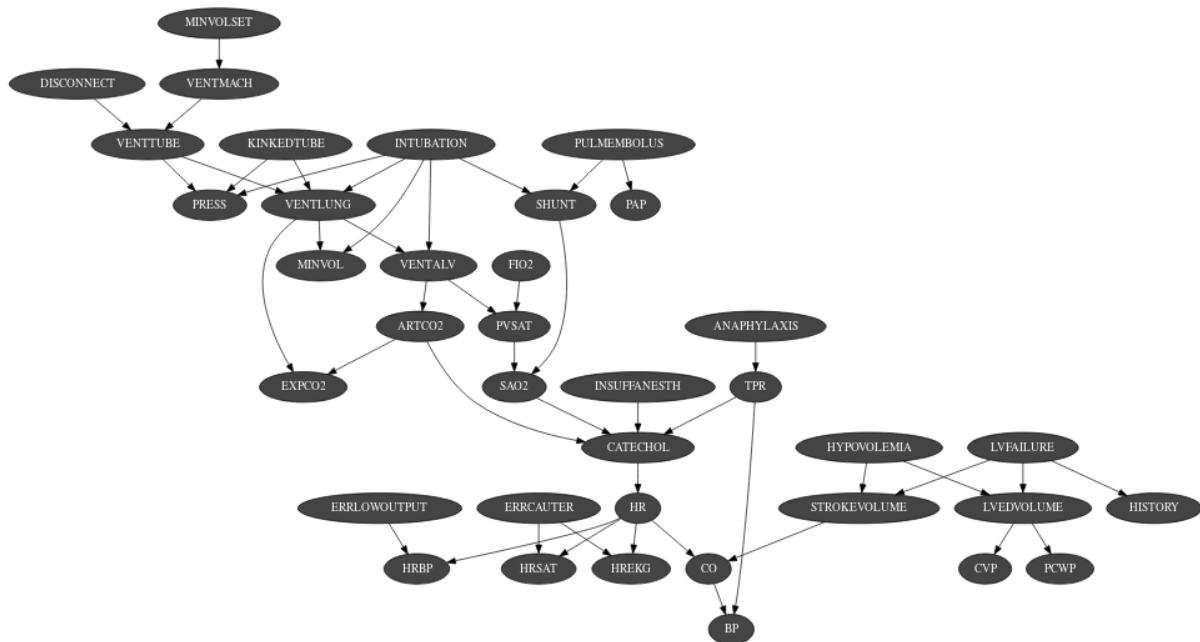
**Returns**

a list containing the sequence of variables

**Return type**

list

## 1.4 Bayesian network



The Bayesian network is the main graphical model of pyAgrim. A Bayesian network is a directed probabilistic graphical model based on a DAG. It represents a joint distribution over a set of random variables. In pyAgrim, the variables are (for now) only discrete.

A Bayesian network uses a directed acyclic graph (DAG) to represent conditional independence in the joint distribution. These conditional independence allow to factorize the joint distribution, thereby allowing to compactly represent very large ones.

$$P(X_1, \dots, X_n) = \prod_{i=1}^n P(X_i | \text{Parents}(X_i))$$

Moreover, inference algorithms can also use this graph to speed up the computations. Finally, the Bayesian networks can be learnt from data.

### Tutorial

- [Tutorials on Bayesian network](#)

### Reference

### 1.4.1 Model

**class** pyAgrum.**BayesNet**(\*args)

BayesNet represents a Bayesian network.

**BayesNet**(name='') -> **BayesNet**

**Parameters:**

- **name** (*str*) – the name of the Bayes Net

**BayesNet**(source) -> **BayesNet**

**Parameters:**

- **source** (*pyAgrum.BayesNet*) – the Bayesian network to copy

**add**(\*args)

Add a variable to the pyAgrum.BayesNet.

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable added
- **descr** (*str*) – the description of the variable (following *fast syntax* (page 279))
- **nbrmod** (*int*) – the number of modalities for the new variable
- **id** (*int*) – the variable forced id in the pyAgrum.BayesNet

**Returns**

the id of the new node

**Return type**

int

**Raises**

- **pyAgrum.DuplicateLabel** (page 287) – If variable.name() or id is already used in this pyAgrum.BayesNet.
- **pyAgrum.NotAllowed** – If nbrmod is less than 2

**addAMPLITUDE**(var)

Others aggregators

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable to be added
- **var** (*DiscreteVariable* (page 25)) –

**Returns**

the id of the added value

**Return type**

int

**addAND**(var)

Add a variable, it's associate node and an AND implementation.

The id of the new variable is automatically generated.

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – The variable added by copy.
- **var** (*DiscreteVariable* (page 25)) –

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

**pyAgrum.SizeError** (page 290) – If variable.domainSize()>2

**addArc**(\*args)

Add an arc in the BN, and update arc.head's CPT.

**Parameters**

- **head** (*Union[int, str]*) – a variable's id (int) or name
- **head** – a variable's id (int) or name

**Raises**

- **pyAgrum.InvalidEdge** (page 289) – If arc.tail and/or arc.head are not in the BN.
- **pyAgrum.DuplicateElement** (page 287) – If the arc already exists.

**Return type**

None

**addArcs**(*listArcs*)

add a list of arcs in te model.

**Parameters****listArcs** (*List[Tuple[intstr, intstr]]*) – the list of arcs**addCOUNT**(*var, value=1*)

Others aggregators

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable to be added
- **var** (*DiscreteVariable* (page 25)) –
- **value** (int) –

**Returns**

the id of the added value

**Return type**

int

**addEXISTS**(*var, value=1*)

Others aggregators

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable to be added
- **var** (*DiscreteVariable* (page 25)) –
- **value** (int) –

**Returns**

the id of the added value

**Return type**

int

**addFORALL**(*var, value=1*)

Others aggregators

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable to be added
- **var** (*DiscreteVariable* (page 25)) –
- **value** (int) –

**Returns**

the id of the added variable.

**Return type**

int

**addLogit**(*\*args*)

Add a variable, its associate node and a Logit implementation.

(The id of the new variable can be automatically generated.)

**Parameters**

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – The variable added by copy
- **externalWeight** (*float*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If id is already used

**addMAX(*var*)**

Others aggregators

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the variable to be added
- **var** ([\*DiscreteVariable\*](#) (page 25)) –

**Returns**

the id of the added value

**Return type**

int

**addMEDIAN(*var*)**

Others aggregators

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the variable to be added
- **var** ([\*DiscreteVariable\*](#) (page 25)) –

**Returns**

the id of the added value

**Return type**

int

**addMIN(*var*)**

Others aggregators

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the variable to be added
- **var** ([\*DiscreteVariable\*](#) (page 25)) –

**Returns**

the id of the added value

**Return type**

int

**addNoisyAND(\**args*)**

Add a variable, its associate node and a noisyAND implementation.

(The id of the new variable can be automatically generated.)

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – The variable added by copy
- **externalWeight** (*float*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If id is already used

**addNoisyOR(\**args*)**

Add a variable, it's associate node and a noisyOR implementation.

Since it seems that the 'classical' noisyOR is the Compound noisyOR, we keep the addNoisyOR as an alias for addNoisyORCompound.

(The id of the new variable can be automatically generated.)

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable added by copy
- **externalWeight** (*float*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[pyAgrum.DuplicateElement](#) (page 287) – If id is already used

**addNoisyORCompound(\*args)**

Add a variable, it's associate node and a noisyOR implementation.

Since it seems that the 'classical' noisyOR is the Compound noisyOR, we keep the addNoisyOR as an alias for addNoisyORCompound.

(The id of the new variable can be automatically generated.)

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable added by copy
- **externalWeight** (*float*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[pyAgrum.DuplicateElement](#) (page 287) – If id is already used

**addNoisyORNet(\*args)**

Add a variable, its associate node and a noisyOR implementation.

Since it seems that the 'classical' noisyOR is the Compound noisyOR, we keep the addNoisyOR as an alias for addNoisyORCompound.

(The id of the new variable can be automatically generated.)

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable added by copy
- **externalWeight** (*float*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

**Returns**

the id of the added variable.

**Return type**

int

**addOR(var)**

Add a variable, it's associate node and an OR implementation.

The id of the new variable is automatically generated.

**Warning:** If parents are not boolean, all value>1 is True

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable added by copy
- **var** ([DiscreteVariable](#) (page 25)) –

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[\*pyAgrum.SizeError\*](#) (page 290) – If variable.domainSize()>2

**addSUM(*var*)**

Others aggregators

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the variable to be added
- **var** ([\*DiscreteVariable\*](#) (page 25)) –

**Returns**

the id of the added value

**Return type**

int

**addStructureListener(*whenNodeAdded=None, whenNodeDeleted=None, whenArcAdded=None, whenArcDeleted=None*)**

Add the listeners in parameters to the list of existing ones.

**Parameters**

- **whenNodeAdded** (*lambda expression*) – a function for when a node is added
- **whenNodeDeleted** (*lambda expression*) – a function for when a node is removed
- **whenArcAdded** (*lambda expression*) – a function for when an arc is added
- **whenArcDeleted** (*lambda expression*) – a function for when an arc is removed

**addVariables(*listFastVariables, default\_nbr\_mod=2*)**

Add a list of variable in the form of ‘fast’ syntax.

**Parameters**

- **listFastVariables** (*List[str]*) – the list of variables in ‘fast’ syntax.
- **default\_nbr\_mod** (*int*) – the number of modalities for the variable if not specified following [\*fast syntax\*](#) (page 279). Note that default\_nbr\_mod=1 is mandatory to create variables with only one modality (for utility for instance).

**Returns**

the list of created ids.

**Return type**

List[int]

**addWeightedArc(\**args*)**

Add an arc in the BN, and update arc.head’s CPT.

**Parameters**

- **head** (*Union[int, str]*) – a variable’s id (int) or name
- **tail** (*Union[int, str]*) – a variable’s id (int) or name
- **causalWeight** (*float*) – the added causal weight

**Raises**

- [\*pyAgrum.InvalidArc\*](#) (page 288) – If arc.tail and/or arc.head are not in the BN.
- [\*pyAgrum.InvalidArc\*](#) (page 288) – If variable in arc.head is not a NoisyOR variable.

**Return type**

None

**ancestors(*norid*)****Parameters**

**norid** (object) –



**Return type**  
object

**arcs()**

**Returns**

The list of arcs in the IBayesNet

**Return type**  
list

**beginTopologyTransformation()**

When inserting/removing arcs, node CPTs change their dimension with a cost in time. begin Multiple Change for all CPTs These functions delay the CPTs change to be done just once at the end of a sequence of topology modification, begins a sequence of insertions/deletions of arcs without changing the dimensions of the CPTs.

**Return type**  
None

**changePotential(\*args)**

change the CPT associated to nodeId to newPot delete the old CPT associated to nodeId.

**Parameters**

- **var** (*Union[int, str]*) – the current name or the id of the variable
- **newPot** (*pyAgrum.Potential* (page 53)) – the new potential

**Raises**

**pyAgrum.NotAllowed** – If newPot has not the same signature as \_\_probaMap[NodeId]

**Return type**  
None

**changeVariableLabel(\*args)**

change the label of the variable associated to nodeId to the new value.

**Parameters**

- **var** (*Union[int, str]*) – the current name or the id of the variable
- **old\_label** (*str*) – the new label
- **new\_label** (*str*) – the new label

**Raises**

**pyAgrum.NotFound** (page 289) – if id/name is not a variable or if old\_label does not exist.

**Return type**  
None

**changeVariableName(\*args)**

Changes a variable's name in the pyAgrum.BayesNet.

This will change the “pyAgrum.DiscreteVariable” names in the pyAgrum.BayesNet.

**Parameters**

- **var** (*Union[int, str]*) – the current name or the id of the variable
- **new\_name** (*str*) – the new name of the variable

**Raises**

- **pyAgrum.DuplicateLabel** (page 287) – If new\_name is already used in this BayesNet.
- **pyAgrum.NotFound** (page 289) – If no variable matches id.

**Return type**  
None

**check()**

**Return type**  
List[str]

**children(norid)**

**Parameters**

- **id** (*int*) – the id of the parent

- **norid** (object) –

**Returns**

the set of all the children

**Return type**

Set

**clear()**

Clear the whole BayesNet

**Return type**

None

**completeInstantiation()****Return type**

[\*Instantiation\*](#) (page 47)

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**cpt(\*args)**

Returns the CPT of a variable.

**Parameters**

**VarId** (*Union[int, str]*) – a variable's id (int) or name

**Returns**

The variable's CPT.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – If no variable's id matches varId.

**dag()****Returns**

a constant reference to the dag of this BayesNet.

**Return type**

[\*pyAgrum.DAG\*](#) (page 8)

**descendants(norid)****Parameters**

**norid** (object) –

**Return type**

object

**dim()**

Returns the dimension (the number of free parameters) in this BayesNet.

**Returns**

the dimension of the BayesNet

**Return type**

int

**empty()**

**Return type**

bool

**endTopologyTransformation()**

Terminates a sequence of insertions/deletions of arcs by adjusting all CPTs dimensions. End Multiple Change for all CPTs.

**Return type***pyAgrum.BayesNet* (page 64)**erase(\*args)**

Remove a variable from the *pyAgrum.BayesNet*.

Removes the corresponding variable from the *pyAgrum.BayesNet* and from all of it's children *pyAgrum.Potential*.

If no variable matches the given id, then nothing is done.

**Parameters**

**var** (*Union[int, str, pyAgrum.DiscreteVariable* (page 25)]) – the current name, the id of the variable or a reference to the variable

**Return type**

None

**eraseArc(\*args)**

Removes an arc in the BN, and update head's CTP.

If (tail, head) doesn't exist, the nothing happens.

**Parameters**

- **arc** (*pyAgrum.Arc* when calling *eraseArc(arc)*) – The arc to be removed.
- **head** (*Union[int, str]*) – a variable's id (int) or name for the head when calling *eraseArc(head,tail)*
- **tail** (*Union[int, str]*) – a variable's id (int) or name for the tail when calling *eraseArc(head,tail)*

**Return type**

None

**exists(node)****Parameters**

**node** (int) –

**Return type**

bool

**existsArc(\*args)****Return type**

bool

**family(norid)****Parameters**

**norid** (object) –

**Return type**

object

**static fastPrototype(dotlike, domainSize=2)**

Create a Bayesian network with a dot-like syntax which specifies:

- the structure 'a->b->c;b->d<-e;'.
- the type of the variables with different syntax:
  - by default, a variable is a *pyAgrum.RangeVariable* using the default domain size (second argument)
  - with 'a[10]', the variable is a *pyAgrum.RangeVariable* using 10 as domain size (from 0 to 9)
  - with 'a[3,7]', the variable is a *pyAgrum.RangeVariable* using a domainSize from 3 to 7

- with ‘a[1,3.14,5,6.2]’, the variable is a `pyAgrum.DiscretizedVariable` using the given ticks (at least 3 values)
- with ‘a{top|middle|bottom}’, the variable is a `pyAgrum.LabelizedVariable` using the given labels.
- with ‘a{-1|5|0|3}’, the variable is a `pyAgrum.IntegerVariable` using the sorted given values.
- with ‘a{-0.5|5.01|0|3.1415}’, the variable is a `pyAgrum.NumericalDiscreteVariable` using the sorted given values.

---

**Note:**

- If the dot-like string contains such a specification more than once for a variable, the first specification will be used.
  - the CPTs are randomly generated.
  - see also `pyAgrum.fastBN`.
- 

## Examples

```
>>> import pyAgrum as gum
>>> bn=pyAgrum.BayesNet.fastPrototype('A->B[1,3]<-C{yes|No}->D[2,4]<-
↳E[1,2.5,3.9]',6)
```

**Parameters**

- **dotlike** (*str*) – the string containing the specification
- **domainSize** (*int*) – the default domain size for variables

**Returns**

the resulting Bayesian network

**Return type**

*pyAgrum.BayesNet* (page 64)

**generateCPT(\*args)**

Randomly generate CPT for a given node in a given structure.

**Parameters**

**node** (*Union[int, str]*) – a variable’s id (int) or name

**Return type**

None

**generateCPTs()**

Randomly generates CPTs for a given structure.

**Return type**

None

**hasSameStructure(other)****Parameters**

**pyAgrum.DAGmodel** – a direct acyclic model

**Returns**

True if all the named node are the same and all the named arcs are the same

**Return type**

bool

**idFromName(name)**

Returns a variable’s id given its name in the graph.

**Parameters**

**name** (*str*) – The variable’s name from which the id is returned.

**Returns**

The variable’s node id.

**Return type**

int

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – If name does not match a variable in the graph

**ids**(*names*)

**isIndependent**(\**args*)

**Return type**

bool

**jointProbability**(*i*)

**Parameters**

**i** (*pyAgrum.instantiation*) – an instantiation of the variables

**Returns**

a parameter of the joint probability for the BayesNet

**Return type**

float

**Warning:** a variable not present in the instantiation is assumed to be instantiated to 0

**loadBIF**(\**args*)

Load a BIF file.

**Parameters**

- **name** (*str*) – the file's name
- **l** (*list*) – list of functions to execute

**Raises**

- [\*pyAgrum.IOError\*](#) (page 288) – If file not found
- [\*pyAgrum.FatalError\*](#) (page 287) – If file is not valid

**Return type**

str

**loadBIFXML**(\**args*)

Load a BIFXML file.

**Parameters**

- **name** (*str*) – the name's file
- **l** (*list*) – list of functions to execute

**Raises**

- [\*pyAgrum.IOError\*](#) (page 288) – If file not found
- [\*pyAgrum.FatalError\*](#) (page 287) – If file is not valid

**Return type**

str

**loadDSL**(\**args*)

Load a DSL file.

**Parameters**

- **name** (*str*) – the file's name
- **l** (*list*) – list of functions to execute

**Raises**

- [\*pyAgrum.IOError\*](#) (page 288) – If file not found
- [\*pyAgrum.FatalError\*](#) (page 287) – If file is not valid

**Return type**

str

**loadNET**(\**args*)

Load a NET file.

**Parameters**

- **name** (*str*) – the name's file
- **l** (*list*) – list of functions to execute

**Raises**

- [\*pyAgrum.IOError\*](#) (page 288) – If file not found

- [`pyAgrum.FatalError`](#) (page 287) – If file is not valid

**Return type**

str

**loadO3PRM(\*args)**

Load an O3PRM file.

**Warning:** The O3PRM language is the only language allowing to manipulate not only DiscretizedVariable but also RangeVariable and LabeledVariable.

**Parameters**

- **name** (str) – the file's name
- **system** (str) – the system's name
- **classpath** (str) – the classpath
- **l** (list) – list of functions to execute

**Raises**

- [`pyAgrum.IOError`](#) (page 288) – If file not found
- [`pyAgrum.FatalError`](#) (page 287) – If file is not valid

**Return type**

str

**loadUAI(\*args)**

Load an UAI file.

**Parameters**

- **name** (str) – the name's file
- **l** (list) – list of functions to execute

**Raises**

- [`pyAgrum.IOError`](#) (page 288) – If file not found
- [`pyAgrum.FatalError`](#) (page 287) – If file is not valid

**Return type**

str

**log10DomainSize()****Return type**

float

**log2JointProbability(i)****Parameters****i** (*pyAgrum.instantiation*) – an instantiation of the variables**Returns**

a parameter of the log joint probability for the BayesNet

**Return type**

float

**Warning:** a variable not present in the instantiation is assumed to be instantiated to 0

**maxNonOneParam()****Returns**

The biggest value (not equal to 1) in the CPTs of the BayesNet

**Return type**

float

**maxParam()****Returns**

the biggest value in the CPTs of the BayesNet

**Return type**

float

**maxVarDomainSize()****Returns**

the biggest domain size among the variables of the BayesNet

**Return type**

int

**minNonZeroParam()****Returns**

the smallest value (not equal to 0) in the CPTs of the IBayesNet

**Return type**

float

**minParam()****Returns**

the smallest value in the CPTs of the IBayesNet

**Return type**

float

**minimalCondSet(\*args)**

Returns, given one or many targets and a list of variables, the minimal set of those needed to calculate the target/targets.

**Parameters**

- **target** (*int*) – The id of the target
- **targets** (*List[int]*) – The ids of the targets
- **list** (*List[int]*) – The list of available variables

**Returns**

The minimal set of variables

**Return type**

Set[int]

**moralGraph(clear=True)**

Returns the moral graph of the BayesNet, formed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected.

**Returns**

The moral graph

**Return type**[pyAgrum.UndiGraph](#) (page 12)**Parameters****clear** (bool) –**moralizedAncestralGraph(nodes)****Parameters****nodes** (object) –**Return type**[UndiGraph](#) (page 12)**names()****Returns**

The names of the graph variables

**Return type**

List[str]

**nodeId(var)****Parameters****var** ([pyAgrum.DiscreteVariable](#) (page 25)) – a variable**Returns**

the id of the variable

**Return type**

int

**Raises**

**pyAgrum.IndexError** – If the graph does not contain the variable

**nodes()****Returns**

the set of ids

**Return type**

Set[int]

**nodeset(names)****Parameters**

**names** (Vector\_string) –

**Return type**

List[int]

**parents(norid)****Parameters**

- **id** – The id of the child node
- **norid** (object) –

**Returns**

the set of the parents ids.

**Return type**

Set

**reverseArc(\*args)**

Reverses an arc while preserving the same joint distribution.

**Parameters**

- **tail** – (int) the id of the tail variable
- **head** – (int) the id of the head variable
- **tail** – (str) the name of the tail variable
- **head** – (str) the name of the head variable
- **arc** ([pyAgrum.Arc](#) (page 3)) – an arc

**Raises**

[pyAgrum.InvalidArc](#) (page 288) – If the arc does not exist or if its reversal would induce a directed cycle.

**Return type**

None

**saveBIF(name)**

Save the BayesNet in a BIF file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**saveBIFXML(name)**

Save the BayesNet in a BIFXML file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**saveDSL(name)**

Save the BayesNet in a DSL file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None



**saveNET(*name*)**

Save the BayesNet in a NET file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**saveO3PRM(*name*)**

Save the BayesNet in an O3PRM file.

**Warning:** The O3PRM language is the only language allowing to manipulate not only DiscretizedVariable but also RangeVariable and LabelizedVariable.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**saveUAI(*name*)**

Save the BayesNet in an UAI file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**size()****Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()****Returns**

the number of arcs in the graph

**Return type**

int

**property thisown**

The membership flag

**toDot()****Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder(*clear=True*)****Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**

[\*pyAgrum.InvalidDirectedCycle\*](#) (page 288) – If this graph contains cycles

**Parameters**

**clear** (bool) –

**variable(\**args*)****Parameters**

- **id** (*int*) – a variable's id

- **name** (*str*) – a variable's name

**Returns**

the variable

**Return type**[pyAgrum.DiscreteVariable](#) (page 25)**Raises****pyAgrum.IndexError** – If the graph does not contain the variable**variableFromName**(*name*)**Parameters****name** (*str*) – a variable's name**Returns**

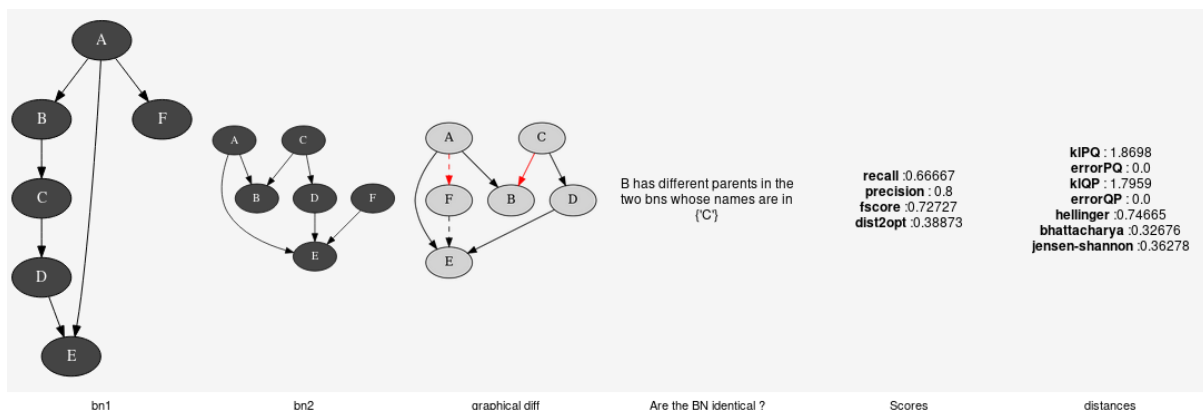
the variable

**Return type**[pyAgrum.DiscreteVariable](#) (page 25)**Raises****pyAgrum.IndexError** – If the graph does not contain the variable**variableNodeMap**()**Returns**

the variable node map

**Return type**[pyAgrum.variableNodeMap](#)

## 1.4.2 Tools for Bayesian networks



aGrUM/pyAgrum provide a set of classes and functions in order to easily work with Bayesian networks.

### Generation of database

**class** [pyAgrum.BNDatabaseGenerator](#)(*bn*)

BNDatabaseGenerator is used to easily generate databases from a [pyAgrum.BayesNet](#).

**Parameters****bn** ([pyAgrum.BayesNet](#) (page 64)) – the Bayesian network used to generate data.**database**()**drawSamples**(*nbSamples*)**Parameters****nbSamples** (int) –**Return type**

float

**log2likelihood()**

**Return type**  
float

**samplesAt(*row, col*)**

**Parameters**

- **row** (int) –
- **col** (int) –

**Return type**  
int

**samplesLabelAt(*row, col*)**

**Parameters**

- **row** (int) –
- **col** (int) –

**Return type**  
str

**samplesNbCols()**

**Return type**  
int

**samplesNbRows()**

**Return type**  
int

**setAntiTopologicalVarOrder()**

**Return type**  
None

**setRandomVarOrder()**

**Return type**  
None

**setTopologicalVarOrder()**

**Return type**  
None

**setVarOrder(\**args*)**

**Return type**  
None

**setVarOrderFromCSV(\**args*)**

**Return type**  
None

**toCSV(\**args*)**

**Return type**  
None

**toDatabaseTable(*useLabels=True*)**

**to\_pandas(*with\_labels=True*)**

export the samples as a pandas.DataFrame.

**Parameters**

**with\_labels** (*bool*) – is the DataFrame full of labels of variables or full of index of labels of variables

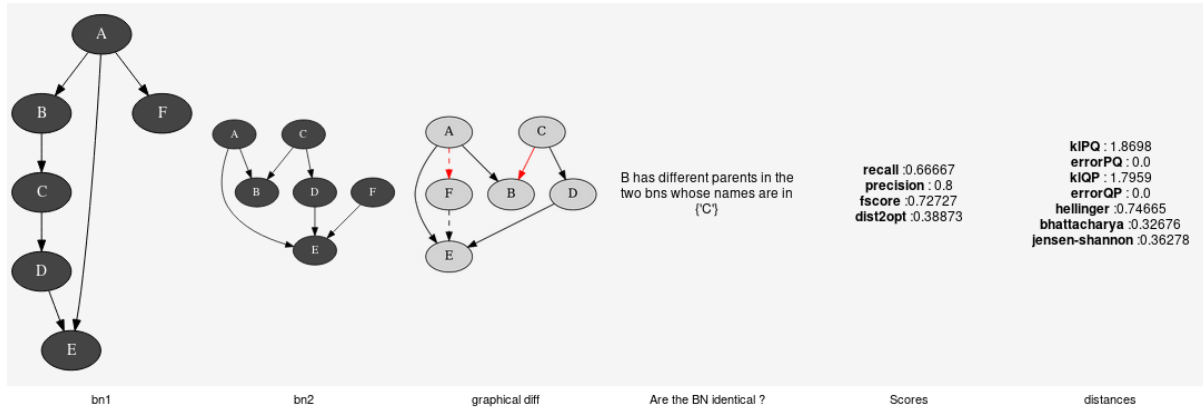
**varOrder()**

**Return type**  
object

```
varOrderNames()
```

**Return type**  
List[str]

## Comparison of Bayesian networks



To compare Bayesian network, one can compare the structure of the BNs (see `pyAgrum.lib.bn_vs_vb.GraphicalBNComparator`). However BNs can also be compared as probability distributions.

```
class pyAgrum.ExactBNdistance(*args)
```

Class representing exact computation of divergence and distance between BNs

**ExactBNdistance(P,Q) -> ExactBNdistance**

**Parameters:**

- **P** (*pyAgrum.BayesNet*) a Bayesian network
- **Q** (*pyAgrum.BayesNet*) another Bayesian network to compare with the first one

**ExactBNdistance(ebnd) -> ExactBNdistance**

**Parameters:**

- **ebnd** (*pyAgrum.ExactBNdistance*) the exact BNdistance to copy

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the 2BNs have not the same domain size of compatible node sets

```
compute()
```

**Returns**

a dictionary containing the different values after the computation.

**Return type**

Dict[str,float]

```
class pyAgrum.GibbsBNdistance(*args)
```

Class representing a Gibbs-Approximated computation of divergence and distance between BNs

**GibbsBNdistance(P,Q) -> GibbsBNdistance**

**Parameters:**

- **P** (*pyAgrum.BayesNet*) – a Bayesian network
- **Q** (*pyAgrum.BayesNet*) – another Bayesian network to compare with the first one

**GibbsBNdistance(gbnd) -> GibbsBNdistance**

**Parameters:**

- **gbnd** (*pyAgrum.GibbsBNdistance*) – the Gibbs BNdistance to copy

**Raises**

*pyAgrum.OperationNotAllowed* (page 290) – If the 2BNs have not the same domain size of compatible node sets

**burnIn()****Returns**

size of burn in on number of iteration

**Return type**

int

**compute()****Returns**

a dictionary containing the different values after the computation.

**Return type**

Dict[str,float]

**continueApproximationScheme(*error*)**

Continue the approximation scheme.

**Parameters**

**error** (*float*) –

**Return type**

bool

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**disableEpsilon()**

Disable epsilon as a stopping criterion.

**Return type**

None

**disableMaxIter()**

Disable max iterations as a stopping criterion.

**Return type**

None

**disableMaxTime()**

Disable max time as a stopping criterion.

**Return type**

None

**disableMinEpsilonRate()**

Disable a min epsilon rate as a stopping criterion.

**Return type**

None

**enableEpsilon()**

Enable epsilon as a stopping criterion.

**Return type**

None

**enableMaxIter()**

Enable max iterations as a stopping criterion.

**Return type**

None

**enableMaxTime()**

Enable max time as a stopping criterion.

**Return type**

None

**enableMinEpsilonRate()**

Enable a min epsilon rate as a stopping criterion.

**Return type**

None

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false**initApproximationScheme()**

Initiate the approximation scheme.

**Return type**

None

**isDrawnAtRandom()****Returns**

True if variables are drawn at random

**Return type**

bool

**isEnabledEpsilon()****Returns**

True if epsilon is used as a stopping criterion.

**Return type**

bool

**isEnabledMaxIter()****Returns**

True if max iterations is used as a stopping criterion

**Return type**

bool

**isEnabledMaxTime()****Returns**

True if max time is used as a stopping criterion

**Return type**

bool

**isEnabledMinEpsilonRate()****Returns**

True if epsilon rate is used as a stopping criterion

**Return type**

bool

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrDrawnVar()****Returns**

the number of variable drawn at each iteration

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **remainingBurnIn()****Returns**

the number of remaining burn in

**Return type**

int

**setBurnIn(*b*)****Parameters****b** (*int*) – size of burn in on number of iteration**Return type**

None

**setDrawnAtRandom(*\_atRandom*)**

**Parameters**

**\_atRandom** (*bool*) – indicates if variables should be drawn at random

**Return type**

None

**setEpsilon(*eps*)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `eps<0`

**Return type**

None

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `max <= 1`

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `timeout<=0.0`

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setNbrDrawnVar(*\_nbr*)****Parameters**

**\_nbr** (*int*) – the number of variables to be drawn at each iteration

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `p<1`

**Return type**

None

**setVerbosity(*v*)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**startOfPeriod()****Returns**

True if it is a start of a period

**Return type**

bool



**stateApproximationScheme()**

**Returns**

the state of the approximation scheme

**Return type**

int

**stopApproximationScheme()**

Stop the approximation scheme.

**Return type**

None

**updateApproximationScheme(*incr=1*)**

Update the approximation scheme.

**Parameters**

**incr** (int) –

**Return type**

None

**verbosity()**

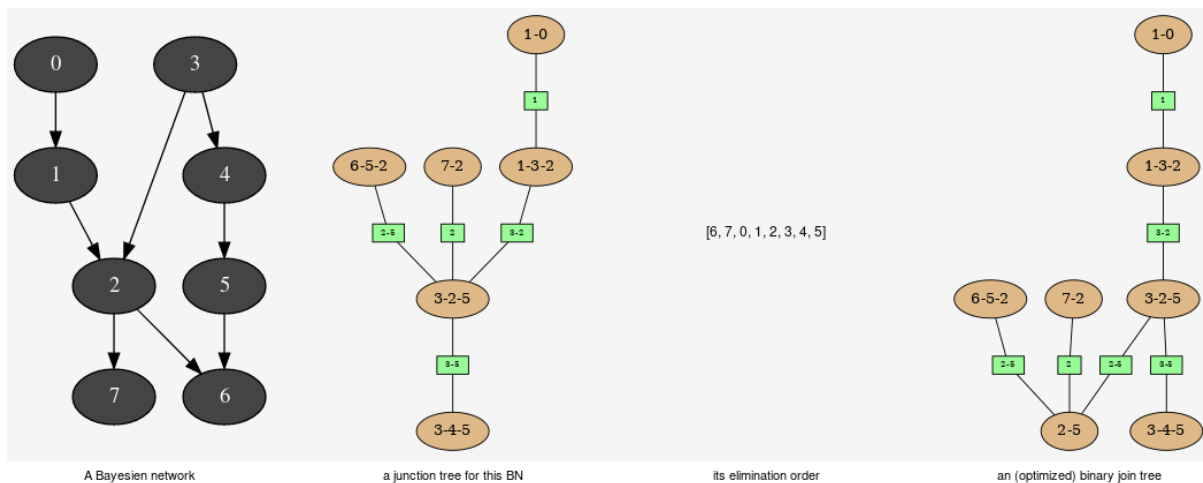
**Returns**

True if the verbosity is enabled

**Return type**

bool

## Explanation and analysis



This tools aimed to provide some different views on the Bayesian network in order to explore its qualitative and/or quantitative behaviours.

**class pyAgrum.JunctionTreeGenerator**

JunctionTreeGenerator is use to generate junction tree or binary junction tree from Bayesian networks.

**JunctionTreeGenerator() -> JunctionTreeGenerator**

default constructor

**binaryJoinTree(\*args)**

Computes the binary joint tree for its parameters. If the first parameter is a graph, the heuristics assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

**Parameters**

- **g** ([pyAgrum.UndiGraph](#) (page 12)) – a undirected graph
- **dag** ([pyAgrum.DAG](#) (page 8)) – a dag

- **bn** ([pyAgrum.BayesNet](#) (page 64)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns**

the current binary joint tree

**Return type**

[pyAgrum.CliqueGraph](#) (page 15)

**eliminationOrder(\*args)**

Computes the elimination for its parameters. If the first parameter is a graph, the heuristics assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

**Parameters**

- **g** ([pyAgrum.UndiGraph](#) (page 12)) – a undirected graph
- **dag** ([pyAgrum.DAG](#) (page 8)) – a dag
- **bn** ([pyAgrum.BayesNet](#) (page 64)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns**

the current elimination order.

**Return type**

[pyAgrum.CliqueGraph](#) (page 15)

**junctionTree(\*args)**

Computes the junction tree for its parameters. If the first parameter is a graph, the heuristics assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

**Parameters**

- **g** ([pyAgrum.UndiGraph](#) (page 12)) – a undirected graph
- **dag** ([pyAgrum.DAG](#) (page 8)) – a dag
- **bn** ([pyAgrum.BayesNet](#) (page 64)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns**

the current junction tree.

**Return type**

[pyAgrum.CliqueGraph](#) (page 15)

**class pyAgrum.EssentialGraph(\*args)**

Class building the essential graph from a BN.

Essential graph is a mixed graph (Chain Graph) that represents the class of markov equivalent Bayesian networks (with the same independency model).

**EssentialGraph(m) -> EssentialGraph****Parameters:**

- **m** ([pyAgrum.DAGmodel](#)) – a DAGmodel

**arcs()****Returns**

The list of arcs in the EssentialGraph

**Return type**

list

**children(id)****Parameters**

**id** (*int*) – the id of the parent

**Returns**

the set of all the children

**Return type**

Set

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**edges()****Returns**

the list of the edges

**Return type**

List

**mixedGraph()****Returns**

the mixed graph

**Return type**

[\*pyAgrum.MixedGraph\*](#) (page 20)

**neighbours(id)****Parameters**

**id** (*int*) – the id of the checked node

**Returns**

The set of edges adjacent to the given node

**Return type**

Set

**nodes()****Return type**

object

**parents(id)****Parameters**

**id** (*int*) – The id of the child node

**Returns**

the set of the parents ids.

**Return type**

Set

**size()****Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()****Returns**

the number of arcs in the graph

**Return type**

int

**sizeEdges()****Returns**

the number of edges in the graph

**Return type**

int

**sizeNodes()****Returns**

the number of nodes in the graph

**Return type**

int

**skeleton()****Return type**[UndiGraph](#) (page 12)**toDot()****Returns**

a friendly display of the graph in DOT format

**Return type**

str

**class** pyAgrum.**MarkovBlanket**(\*args)

Class building the Markov blanket of a node in a graph.

**MarkovBlanket(m,n) -> MarkovBlanket****Parameters:**

- **m** (*pyAgrum.DAGmodel*) – a DAGmodel
- **n** (int) – a node id

**MarkovBlanket(m,name) -> MarkovBlanket****Parameters:**

- **m** (*pyAgrum.DAGmodel*) – a DAGmodel
- **name** (*str*) – a node name

**arcs()****Returns**

the list of the arcs

**Return type**

List

**children(id)****Parameters****id** (*int*) – the id of the parent**Returns**

the set of all the children

**Return type**

Set

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**dag()**

**Returns**

a copy of the DAG

**Return type**

*pyAgrum.DAG* (page 8)

**hasSameStructure(*other*)**

**Parameters**

**pyAgrum.DAGmodel** – a direct acyclic model

**Returns**

True if all the named node are the same and all the named arcs are the same

**Return type**

bool

**nodes()**

**Returns**

the set of ids

**Return type**

set

**parents(*id*)**

**Parameters**

**id** (int) – The id of the child node

**Returns**

the set of the parents ids.

**Return type**

Set

**size()**

**Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs()**

**Returns**

the number of arcs in the graph

**Return type**

int

**sizeNodes()**

**Returns**

the number of nodes in the graph

**Return type**

int

**toDot()**

**Returns**

a friendly display of the graph in DOT format

**Return type**

str

## Fragment of Bayesian networks

This class proposes a shallow copy of a part of Bayesian network. It can be used as a Bayesian network for inference algorithms (for instance).

**class** pyAgrum.**BayesNetFragment**(*bn*)

BayesNetFragment represents a part of a Bayesian network (subset of nodes). By default, the arcs and the CPTs are the same as the BN but local CPTs can be build to express different local dependencies. All the non local CPTs are not copied. Therefore a BayesNetFragment is a light object.

**BayesNetFragment**(BayesNet *bn*) -> **BayesNetFragment**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – the bn referred by the fragment

**Parameters**

**bn** (*IBayesNet*) –

**addArcs**(*listArcs*)

add a list of arcs in te model.

**Parameters**

**listArcs** (*List[Tuple[intstr, intstr]]*) – the list of arcs

**addStructureListener**(*whenNodeAdded=None, whenNodeDeleted=None, whenArcAdded=None, whenArcDeleted=None*)

Add the listeners in parameters to the list of existing ones.

**Parameters**

- **whenNodeAdded** (*lambda expression*) – a function for when a node is added
- **whenNodeDeleted** (*lambda expression*) – a function for when a node is removed
- **whenArcAdded** (*lambda expression*) – a function for when an arc is added
- **whenArcDeleted** (*lambda expression*) – a function for when an arc is removed

**addVariables**(*listFastVariables, default\_nbr\_mod=2*)

Add a list of variable in the form of ‘fast’ syntax.

**Parameters**

- **listFastVariables** (*List[str]*) – the list of variables in ‘fast’ syntax.
- **default\_nbr\_mod** (*int*) – the number of modalities for the variable if not specified following *fast syntax* (page 279). Note that default\_nbr\_mod=1 is mandatory to create variables with only one modality (for utility for instance).

**Returns**

the list of created ids.

**Return type**

List[int]

**ancestors**(*norid*)

**Parameters**

**norid** (object) –

**Return type**

object

**arcs**()

**Returns**

The list of arcs in the IBayesNet

**Return type**

list

**check()**

**Return type**  
List[str]

**checkConsistency(\*args)**

If a variable is added to the fragment but not its parents, there is no CPT constant for this variable. This function checks the consistency for a variable of for all.

**Parameters**

**n** (*int*, *str* (*optional*)) – the id or the name of the variable. If no argument, the function checks all the variables.

**Returns**

True if the variable(s) is consistent.

**Return type**

boolean

**Raises**

- [\*pyAgrum.NotFound\*](#) (page 289) –
- **if the node is not found.** –

**children(*norid*)**

**Parameters**

- **id** (*int*) – the id of the parent
- **norid** (*object*) –

**Returns**

the set of all the children

**Return type**

Set

**completeInstantiation()**

**Return type**

[\*Instantiation\*](#) (page 47)

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (*int*)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**cpt(\*args)**

Returns the CPT of a variable.

**Parameters**

- **VarId** (*int*) – A variable's id in the pyAgrum.IBayesNet.
- **name** (*str*) – A variable's name in the pyAgrum.IBayesNet.

**Returns**

The variable's CPT.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – If no variable's id matches varId.

**dag()**

**Returns**

a constant reference to the dag of this BayesNet.

**Return type***pyAgrum.DAG* (page 8)**descendants**(*norid*)**Parameters****norid** (object) –**Return type**

object

**dim**()

Returns the dimension (the number of free parameters) in this BayesNet.

**Returns**

the dimension of the BayesNet

**Return type**

int

**empty**()**Return type**

bool

**exists**(*node*)**Parameters****node** (int) –**Return type**

bool

**existsArc**(\**args*)**Return type**

bool

**family**(*norid*)**Parameters****norid** (object) –**Return type**

object

**hasSameStructure**(*other*)**Parameters****pyAgrum.DAGmodel** – a direct acyclic model**Returns**

True if all the named node are the same and all the named arcs are the same

**Return type**

bool

**idFromName**(*name*)

Returns a variable's id given its name in the graph.

**Parameters****name** (*str*) – The variable's name from which the id is returned.**Returns**

The variable's node id.

**Return type**

int

**Raises***pyAgrum.NotFound* (page 289) – If name does not match a variable in the graph**ids**(*names*)**installAscendants**(\**args*)

Add the variable and all its ascendants in the fragment. No inconsistent node are created.

**Parameters****n** (*int*, *str*) – the id or the name of the variable.



**Raises**

- [`pyAgrum.NotFound`](#) (page 289) –
- if the node is not found. –

**Return type**

None

**installCPT(\*args)**

Install a local CPT for a node. Doing so, it changes the parents of the node in the fragment.

**Parameters**

- **n** (*int*, *str*) – the id or the name of the variable.
- **pot** ([`Potential`](#) (page 53)) – the Potential to install

**Raises**

[`pyAgrum.NotFound`](#) (page 289) – if the node is not found.

**Return type**

None

**installMarginal(\*args)**

Install a local marginal for a node. Doing so, it removes the parents of the node in the fragment.

**Parameters**

- **n** (*int*, *str*) – the id or the name of the variable.
- **pot** ([`Potential`](#) (page 53)) – the Potential (marginal) to install

**Raises**

[`pyAgrum.NotFound`](#) (page 289) – if the node is not found.

**Return type**

None

**installNode(\*args)**

Add a node to the fragment. The arcs that can be added between installed nodes are created. No specific CPT are created. Then either the parents of the node are already in the fragment and the node is consistent, or the parents are not in the fragment and the node is not consistent.

**Parameters**

**n** (*int*, *str*) – the id or the name of the variable.

**Raises**

[`pyAgrum.NotFound`](#) (page 289) – if the node is not found.

**Return type**

None

**isIndependent(\*args)****Return type**

bool

**isInstalledNode(\*args)**

Check if a node is in the fragment

**Parameters**

**n** (*int*, *str*) – the id or the name of the variable.

**Return type**

bool

**jointProbability(i)****Parameters**

**i** (*pyAgrum.instantiation*) – an instantiation of the variables

**Returns**

a parameter of the joint probability for the BayesNet

**Return type**

float

**Warning:** a variable not present in the instantiation is assumed to be instantiated to 0

**log10DomainSize()****Return type**  
float**log2JointProbability(*i*)****Parameters***i* (*pyAgrum.instantiation*) – an instantiation of the variables**Returns**

a parameter of the log joint probability for the BayesNet

**Return type**  
float

**Warning:** a variable not present in the instantiation is assumed to be instantiated to 0

**maxNonOneParam()****Returns**

The biggest value (not equal to 1) in the CPTs of the BayesNet

**Return type**  
float**maxParam()****Returns**

the biggest value in the CPTs of the BayesNet

**Return type**  
float**maxVarDomainSize()****Returns**

the biggest domain size among the variables of the BayesNet

**Return type**  
int**minNonZeroParam()****Returns**

the smallest value (not equal to 0) in the CPTs of the IBayesNet

**Return type**  
float**minParam()****Returns**

the smallest value in the CPTs of the IBayesNet

**Return type**  
float**minimalCondSet(\*args)**

Returns, given one or many targets and a list of variables, the minimal set of those needed to calculate the target/targets.

**Parameters**

- **target** (*int*) – The id of the target
- **targets** (*List[int]*) – The ids of the targets
- **list** (*List[int]*) – The list of available variables

**Returns**

The minimal set of variables

**Return type**  
Set[int]

**moralGraph**(*clear=True*)

Returns the moral graph of the BayesNet, formed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected.

**Returns**

The moral graph

**Return type**

[\*pyAgrum.UndiGraph\*](#) (page 12)

**Parameters**

**clear** (bool) –

**moralizedAncestralGraph**(*nodes*)

**Parameters**

**nodes** (object) –

**Return type**

[\*UndiGraph\*](#) (page 12)

**names**()

**Returns**

The names of the graph variables

**Return type**

List[str]

**nodeId**(*var*)

**Parameters**

**var** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – a variable

**Returns**

the id of the variable

**Return type**

int

**Raises**

**pyAgrum.IndexError** – If the graph does not contain the variable

**nodes**()

**Returns**

the set of ids

**Return type**

Set[int]

**nodeset**(*names*)

**Parameters**

**names** (Vector\_string) –

**Return type**

List[int]

**parents**(*norid*)

**Parameters**

- **id** – The id of the child node
- **norid** (object) –

**Returns**

the set of the parents ids.

**Return type**

Set

**property**(*name*)

**Parameters**

**name** (str) –

**Return type**

str

**propertyWithDefault**(*name*, *byDefault*)

**Parameters**

- **name** (str) –
- **byDefault** (str) –

**Return type**

str

**setProperty**(*name*, *value*)

**Parameters**

- **name** (str) –
- **value** (str) –

**Return type**

None

**size**()

**Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs**()

**Returns**

the number of arcs in the graph

**Return type**

int

**toBN**()

Create a BayesNet from a fragment.

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – if the fragment is not consistent.

**Return type**

[\*BayesNet\*](#) (page 64)

**toDot**()

**Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder**(*clear=True*)

**Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**

[\*pyAgrum.InvalidDirectedCycle\*](#) (page 288) – If this graph contains cycles

**Parameters**

**clear** (bool) –

**uninstallCPT**(\**args*)

Remove a local CPT. The fragment can become inconsistent.

**Parameters**

**n** (*int*, *str*) – the id or the name of the variable.

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – if the node is not found.

**Return type**

None

**uninstallNode(\*args)**

Remove a node from the fragment. The fragment can become inconsistent.

**Parameters**

**n** (*int*, *str*) – the id or the name of the variable.

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – if the node is not found.

**Return type**

None

**variable(\*args)****Parameters**

- **id** (*int*) – a variable's id
- **name** (*str*) – a variable's name

**Returns**

the variable

**Return type**

[\*pyAgrum.DiscreteVariable\*](#) (page 25)

**Raises**

**pyAgrum.IndexError** – If the graph does not contain the variable

**variableFromName(name)****Parameters**

**name** (*str*) – a variable's name

**Returns**

the variable

**Return type**

[\*pyAgrum.DiscreteVariable\*](#) (page 25)

**Raises**

**pyAgrum.IndexError** – If the graph does not contain the variable

**variableNodeMap()****Returns**

the variable node map

**Return type**

`pyAgrum.variableNodeMap`

**whenArcAdded(src, \_from, to)****Parameters**

- **src** (object) –
- **\_from** (int) –
- **to** (int) –

**Return type**

None

**whenArcDeleted(src, \_from, to)****Parameters**

- **src** (object) –
- **\_from** (int) –
- **to** (int) –

**Return type**

None

**whenNodeAdded(src, id)****Parameters**

- **src** (object) –
- **id** (int) –

**Return type**

None

**whenNodeDeleted**(*src, id*)

**Parameters**

- **src** (object) –
- **id** (int) –

**Return type**

None

### 1.4.3 Inference

Inference is the process that consists in computing new probabilistic information from a Bayesian network and some evidence. aGrUM/pyAgrum mainly focus and the computation of (joint) posterior for some variables of the Bayesian networks given soft or hard evidence that are the form of likelihoods on some variables. Inference is a hard task (NP-complete). aGrUM/pyAgrum implements exact inference but also approximated inference that can converge slowly and (even) not exactly but that can in many cases be useful for applications.

### 1.4.4 Exact Inference

#### Lazy Propagation

Lazy Propagation is the main exact inference for classical Bayesian networks in aGrUM/pyAgrum.

**class** pyAgrum.LazyPropagation(\*args)

Class used for Lazy Propagation

**LazyPropagation(bn) -> LazyPropagation**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**BN()**

**Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shannon's entropy of a node given the observation

**Return type**

float

**I(\*args)**

**Parameters**

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

**Returns**

- ----- –
- **float** – the Mutual Information of X and Y given the observation

**Return type**

float

**VI(\*args)**

**Parameters**

- **X** (*int* or *str*) – a node Id or a node name
- **Y** (*int* or *str*) – another node Id or node name

Returns

- -----
- **float** – variation of information between X and Y

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addJointTarget(targets)**

Add a list of nodes as a new joint target. As a collateral effect, every node is added as a marginal target.

**Parameters**

- **list** – a list of names of nodes
- **targets** (object) –

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If some node(s) do not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllJointTargets()**

Clear all previously defined joint targets.

**Return type**

None

**eraseAllMarginalTargets()**

Clear all the previously defined marginal targets.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- [\*pyAgrum.IndexError\*](#) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseJointTarget(targets)**

Remove, if existing, the joint target.

**Parameters**

- **list** – a list of names or Ids of nodes
- **targets** (object) –

**Raises**

- [\*pyAgrum.IndexError\*](#) – If one of the node does not belong to the Bayesian network



- [`pyAgrum.UndefinedElement`](#) (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- [`pyAgrum.UndefinedElement`](#) (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**Create a `pyAgrum.Potential` for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**a Potential for  $P(\text{targets}|\text{evs})$ **Return type**[`pyAgrum.Potential`](#) (page 53)**evidenceJointImpact(\*args)**Create a `pyAgrum.Potential` for  $P(\text{joint targets}|\text{evs})$  (for all instantiation of targets and evs)**Parameters**

- **targets** (*List[intstr]*) – a list of node Ids or node names
- **evs** (*Set[intstr]*) – a set of nodes ids or names.

**Returns**a Potential for  $P(\text{target}|\text{evs})$ **Return type**[`pyAgrum.Potential`](#) (page 53)**Raises**

**pyAgrum.Exception** – If some evidence entered into the Bayes net are incompatible (their joint proba = 0)

**evidenceProbability()****Returns**

the probability of evidence

**Return type**

float

**getNumberOfThreads()**

returns the number of threads used by LazyPropagation during inferences.

**Returns**

the number of threads used by LazyPropagation during inferences

**Return type**

int

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**isGumNumberOfThreadsOverriden()**

Indicates whether LazyPropagation currently overrides aGrUM's default number of threads (see method `setNumberOfThreads`).

**Returns**

A Boolean indicating whether LazyPropagation currently overrides aGrUM's default number of threads

**Return type**

bool

**isJointTarget(targets)****Parameters**

- **list** – a list of nodes ids or names.
- **targets** (object) –

**Returns**

True if target is a joint target.

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**joinTree()****Returns**

the current join tree used

**Return type**

*pyAgrum.CliqueGraph* (page 15)

**jointMutualInformation(targets)****Parameters**

**targets** (object) –

**Return type**

float

**jointPosterior(targets)**

Compute the joint posterior of a set of nodes.

**Parameters**

**list** – the list of nodes whose posterior joint probability is wanted

**Warning:** The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used by the Potential.

**Returns**

a const ref to the posterior joint probability of the set of nodes.

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

**pyAgrum.UndefinedElement** (page 291) – If an element of nodes is not in targets

**Parameters**

**targets** (object) –

**jointTargets()****Returns**

the list of target sets

**Return type**

list

**junctionTree()****Returns**

the current junction tree

**Return type**

*pyAgrum.CliqueGraph* (page 15)

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrJointTargets()****Returns**

the number of joint targets

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network**setMaxMemory(gigabytes)**

sets an upper bound on the memory consumption admissible

**Parameters****gigabytes** (*float*) – this upper bound in gigabytes.

**Return type**

None

**setNumberOfThreads(*nb*)**

If the argument *nb* is different from 0, this number of threads will be used during inferences, hence overriding aGrUM's default number of threads. If, on the contrary, *nb* is equal to 0, the parallelized inference engine will comply with aGrUM's default number of threads.

**Parameters**

**nb** (*int*) – the number of threads to be used by ShaferShenoyMNIInference

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(*evidces*)**

Apply `chgEvidence(key,value)` for every pairs in *evidces* (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**Shafer Shenoy Inference**

**class** `pyAgrum.ShaferShenoyInference(*args)`

Class used for Shafer-Shenoy inferences.

**ShaferShenoyInference(bn) -> ShaferShenoyInference**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**BN()****Returns**

A constant reference over the *IBayesNet* referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)****Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**I(\*args)****Parameters**

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

**Returns**

- -----
- **float** – the Mutual Information of X and Y given the observation

**Return type**

float

**VI(\*args)****Parameters**

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

**Returns**

- -----
- **float** – variation of information between X and Y

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addJointTarget**(*targets*)

Add a list of nodes as a new joint target. As a collateral effect, every node is added as a marginal target.

**Parameters**

- **list** – a list of names of nodes
- **targets** (object) –

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If some node(s) do not belong to the Bayesian network

**Return type**

None

**addTarget**(\**args*)

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence**(\**args*)

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseAllEvidence**()

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllJointTargets**()

Clear all previously defined joint targets.

**Return type**

None

**eraseAllMarginalTargets**()

Clear all the previously defined marginal targets.

**Return type**

None

**eraseAllTargets**()

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseJointTarget(targets)**

Remove, if existing, the joint target.

**Parameters**

- **list** – a list of names or Ids of nodes
- **targets** (object) –

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a pyAgrum.Potential for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for  $P(\text{targets}|\text{evs})$

**Return type**

*pyAgrum.Potential* (page 53)

**evidenceJointImpact(\*args)**

Create a pyAgrum.Potential for  $P(\text{joint targets}|\text{evs})$  (for all instantiation of targets and evs)

**Parameters**

- **targets** (*List[intstr]*) – a list of node Ids or node names
- **evs** (*Set[intstr]*) – a set of nodes ids or names.



**Returns**

a Potential for  $P(\text{target}|\text{evs})$

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

**pyAgrum.Exception** – If some evidence entered into the Bayes net are incompatible (their joint proba = 0)

**evidenceProbability()****Returns**

the probability of evidence

**Return type**

float

**getNumberOfThreads()**

returns the number of threads used by LazyPropagation during inferences.

**Returns**

the number of threads used by LazyPropagation during inferences

**Return type**

int

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**isGumNumberOfThreadsOverriden()**

Indicates whether LazyPropagation currently overrides aGrUM's default number of threads (see method `setNumberOfThreads`).

**Returns**

A Boolean indicating whether LazyPropagation currently overrides aGrUM's default number of threads

**Return type**

bool

**isJointTarget(*targets*)****Parameters**

- **list** – a list of nodes ids or names.
- **targets** (object) –

**Returns**

True if target is a joint target.

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**isTarget(\**args*)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**joinTree()****Returns**

the current join tree used

**Return type**

*pyAgrum.CliqueGraph* (page 15)

**jointMutualInformation(*targets*)****Parameters**

**targets** (object) –

**Return type**

float

**jointPosterior(*targets*)**

Compute the joint posterior of a set of nodes.

**Parameters**

**list** – the list of nodes whose posterior joint probability is wanted

**Warning:** The order of the variables given by the list here or when the `jointTarget` is declared can not be assumed to be used by the `Potential`.

**Returns**

a const ref to the posterior joint probability of the set of nodes.

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**Parameters**

**targets** (object) –

**jointTargets()****Returns**

the list of target sets

**Return type**

list

**junctionTree()****Returns**

the current junction tree

**Return type**

[\*pyAgrum.CliqueGraph\*](#) (page 15)

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what `makeInference` should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrJointTargets()****Returns**

the number of joint targets

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node pyAgrum.InvalidArgument If the size of a value is different from the domain side of the node pyAgrum.FatalError If one value is a vector of 0s pyAgrum.UndefinedElement If one node does not belong to the Bayesian network

**setMaxMemory(gigabytes)**

sets an upper bound on the memory consumption admissible

**Parameters**

**gigabytes** (*float*) – this upper bound in gigabytes.

**Return type**

None

**setNumberOfThreads(nb)**

If the argument nb is different from 0, this number of threads will be used during inferences, hence overriding aGrUM's default number of threads. If, on the contrary, nb is equal to 0, the parallelized inference engine will comply with aGrUM's default number of threads.

**Parameters**

**nb** (*int*) – the number of threads to be used by ShaferShenoyMNIInference

**Return type**

None

**setTargets(targets)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(evidces)**

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If one value is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of a value is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If one value is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If one node does not belong to the Bayesian network

**Variable Elimination**

**class** `pyAgrum.VariableElimination(*args)`

Class used for Variable Elimination inference algorithm.

**VariableElimination(bn) -> VariableElimination**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**BN()**

**Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

`pyAgrum.IBayesNet`

**Raises**

- [`pyAgrum.UndefinedElement`](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node already has an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of vals is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If vals is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addJointTarget**(*targets*)

Add a list of nodes as a new joint target. As a collateral effect, every node is added as a marginal target.

**Parameters**

- **list** – a list of names of nodes
- **targets** (object) –

**Raises**

[`pyAgrum.UndefinedElement`](#) (page 291) – If some node(s) do not belong to the Bayesian network

**Return type**

None

**addTarget**(\**args*)

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[`pyAgrum.UndefinedElement`](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence**(\**args*)

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node does not already have an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of vals is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If vals is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseAllEvidence**()

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets**()

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**Return type**

None

**eraseJointTarget(targets)**

Remove, if existing, the joint target.

**Parameters**

- **list** – a list of names or Ids of nodes
- **targets** (object) –

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for P(targets|evs)

**Return type***pyAgrum.Potential* (page 53)**evidenceJointImpact(targets, evs)**

Create a pyAgrum.Potential for P(joint targets|evs) (for all instantiation of targets and evs)

**Parameters**

- **targets** (*List[intstr]*) – a list of node Ids or node names
- **evs** (*Set[intstr]*) – a set of nodes ids or names.

**Returns**

a Potential for P(target|evs)

**Return type***pyAgrum.Potential* (page 53)**Raises**

**pyAgrum.Exception** – If some evidence entered into the Bayes net are incompatible (their joint proba = 0)

**getNumberOfThreads()**

returns the number of threads used by LazyPropagation during inferences.

**Returns**

the number of threads used by LazyPropagation during inferences

**Return type**

int

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**isGumNumberOfThreadsOverriden()**

Indicates whether LazyPropagation currently overrides aGrUM's default number of threads (see method `setNumberOfThreads`).

**Returns**

A Boolean indicating whether LazyPropagation currently overrides aGrUM's default number of threads

**Return type**

bool



**isJointTarget**(*targets*)**Parameters**

- **list** – a list of nodes ids or names.
- **targets** (object) –

**Returns**

True if target is a joint target.

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**isTarget**(\**args*)**Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**jointMutualInformation**(*targets*)**Parameters**

**targets** (object) –

**Return type**

float

**jointPosterior**(*targets*)

Compute the joint posterior of a set of nodes.

**Parameters**

**list** – the list of nodes whose posterior joint probability is wanted

**Warning:** The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used by the Potential.

**Returns**

a const ref to the posterior joint probability of the set of nodes.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**Parameters**

**targets** (object) –

**jointTargets**()**Returns**

the list of target sets

**Return type**

list

**junctionTree**(*id*)**Returns**

the current junction tree

**Return type***pyAgrum.CliqueGraph* (page 15)**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type***pyAgrum.Potential* (page 53)**Raises**

*pyAgrum.UndefinedElement* (page 291) – If an element of nodes is not in targets

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

*pyAgrum.InvalidArgument* (page 288) – If one value is not a value for the node  
*pyAgrum.InvalidArgument* If the size of a value is different from the domain side of the node  
*pyAgrum.FatalError* If one value is a vector of 0s  
*pyAgrum.UndefinedElement* If one node does not belong to the Bayesian network

**setMaxMemory**(*gigabytes*)

sets an upper bound on the memory consumption admissible

**Parameters**

**gigabytes** (*float*) – this upper bound in gigabytes.

**Return type**

None

**setNumberOfThreads**(*nb*)

If the argument *nb* is different from 0, this number of threads will be used during inferences, hence overriding aGrUM's default number of threads. If, on the contrary, *nb* is equal to 0, the parallelized inference engine will comply with aGrUM's default number of threads.

**Parameters**

**nb** (*int*) – the number of threads to be used by ShaferShenoyMNIInference

**Return type**

None

**setTargets**(*targets*)

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**softEvidenceNodes**()**Returns**

the set of nodes with soft evidence

**Return type**

set

**targets**()**Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence**(*evidces*)

Apply *chgEvidence*(key,value) for every pairs in *evidces* (or *addEvidence*).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

## 1.4.5 Approximated Inference

### Loopy Belief Propagation

**class** pyAgrum.LoopyBeliefPropagation(*bn*)

Class used for inferences using loopy belief propagation algorithm.

**LoopyBeliefPropagation(bn) -> LoopyBeliefPropagation**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**Parameters**

**bn** (*IBayesNet*) –

**BN()**

**Returns**

A constant reference over the *IBayesNet* referenced by this class.

**Return type**

*pyAgrum.IBayesNet*

**Raises**

*pyAgrum.UndefinedElement* (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- *pyAgrum.InvalidArgument* (page 288) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 288) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 288) – If the size of vals is different from the domain side of the node
- *pyAgrum.FatalError* (page 287) – If vals is a vector of 0s
- *pyAgrum.UndefinedElement* (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget**(\*args)

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact**(target, evs)

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for P(targets|evs)

**Return type**

*pyAgrum.Potential* (page 53)

**hardEvidenceNodes**()**Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence**(\*args)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence**(nodeName)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence**(\*args)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history**()**Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget**(\*args)**Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If node Id is not in the Bayesian network

**makeInference**()

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**maxIter**()**Returns**

the criterion on number of iterations

**Return type**

int

**maxTime**()**Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme**()**Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()**

**Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()**

**Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()**

**Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()**

**Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()**

**Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()**

**Returns**

the number of marginal targets

**Return type**

int

**periodSize()**

**Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**setEpsilon(eps)**



**Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `eps<0`

**Return type**

None

**setEvidence(evidces)**

Erase all the evidences and apply `addEvidence(key,value)` for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network

**setMaxIter(max)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `max <= 1`

**Return type**

None

**setMaxTime(timeout)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `timeout<=0.0`

**Return type**

None

**setMinEpsilonRate(rate)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(p)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `p<1`

**Return type**

None

**setTargets(targets)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity(v)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**softEvidenceNodes()**

**Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()**

**Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(evidces)**

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()**

**Returns**

True if the verbosity is enabled

**Return type**

bool

## Sampling

### Gibbs Sampling

**class pyAgrum.GibbsSampling(bn)**

Class for making Gibbs sampling inference in Bayesian networks.

**GibbsSampling(bn) -> GibbsSampling**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**Parameters**

**bn** (*IBayesNet*) –

**BN()**

**Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shannon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**burnIn()**

**Returns**

size of burn in on number of iteration

**Return type**

int

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node does not already have an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of vals is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If vals is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**

[`pyAgrum.Potential`](#) (page 53)

**Raises**

[`UndefinedElement`](#) (page 291) – If an element of nodes is not in targets

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

[`pyAgrum.IndexError`](#) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact**(*target, evs*)Create a pyAgrum.Potential for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**a Potential for  $P(\text{targets}|\text{evs})$ **Return type***pyAgrum.Potential* (page 53)**hardEvidenceNodes**()**Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence**(\**args*)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasHardEvidence**(*nodeName*)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasSoftEvidence**(\**args*)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**history()****Returns**

the scheme history

**Return type**

tuple

**Raises****pyAgrum.OperationNotAllowed** (page 290) – If the scheme did not performed or if verbosity is set to false**isDrawnAtRandom()****Returns**

True if variables are drawn at random

**Return type**

bool

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrDrawnVar()****Returns**

the number of variable drawn at each iteration

**Return type**

int

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**setBurnIn(*b*)****Parameters**

**b** (*int*) – size of burn in on number of iteration

**Return type**

None

**setDrawnAtRandom(*\_atRandom*)****Parameters**

**\_atRandom** (*bool*) – indicates if variables should be drawn at random

**Return type**

None

**setEpsilon(*eps*)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $eps < 0$

**Return type**

None

**setEvidence(*evidces*)**

Erase all the evidences and apply `addEvidence(key,value)` for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $max \leq 1$

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $timeout \leq 0.0$

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setNbrDrawnVar(*\_nbr*)****Parameters**

**\_nbr** (*int*) – the number of variables to be drawn at each iteration



**Return type**

None

**setPeriodSize(*p*)****Parameters****p** (*int*) – number of samples between 2 stopping**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters****targets** (*set*) – a set of targets**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net**setVerbosity(*v*)****Parameters****v** (*bool*) – verbosity**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(*evidces*)**Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).**Parameters****evidces** (*dict*) – a dict of evidences**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()****Returns**

True if the verbosity is enabled

**Return type**

bool

## Monte Carlo Sampling

**class** pyAgrum.MonteCarloSampling(*bn*)

Class used for Monte Carlo sampling inference algorithm.

**MonteCarloSampling(bn) -> MonteCarloSampling**

**Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**Parameters**

**bn** (*IBayesNet*) –

**BN()**

**Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a `pyAgrum.Potential` for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for  $P(\text{targets}|\text{evs})$

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)**

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()**

**Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[pyAgrum.Potential](#) (page 53)

**Raises**

[pyAgrum.UndefinedElement](#) (page 291) – If an element of nodes is not in targets

**setEpsilon(*eps*)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If  $\text{eps} < 0$

**Return type**

None

**setEvidence(*evidces*)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[pyAgrum.InvalidArgument](#) (page 288) – If one value is not a value for the node pyAgrum.InvalidArgument If the size of a value is different from the domain side of the node pyAgrum.FatalError If one value is a vector of 0s pyAgrum.UndefinedElement If one node does not belong to the Bayesian network

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If  $\text{max} \leq 1$

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If  $\text{timeout} \leq 0.0$

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If  $p < 1$

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters****targets** (*set*) – a set of targets**Raises**[pyAgrum.UndefinedElement](#) (page 291) – If one target is not in the Bayes net**setVerbosity(*v*)****Parameters****v** (*bool*) – verbosity**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(*evidces*)**

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**

- [pyAgrum.InvalidArgument](#) (page 288) – If one value is not a value for the node
- [pyAgrum.InvalidArgument](#) (page 288) – If the size of a value is different from the domain side of the node
- [pyAgrum.FatalError](#) (page 287) – If one value is a vector of 0s
- [pyAgrum.UndefinedElement](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()****Returns**

True if the verbosity is enabled

**Return type**

bool

## Weighted Sampling

**class pyAgrum.WeightedSampling(*bn*)**

Class used for Weighted sampling inference algorithm.

**WeightedSampling(*bn*) -> WeightedSampling****Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**Parameters****bn** (*IBayesNet*) –



**BN()****Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)****Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**[\*pyAgrum.IndexError\*](#) – If the node does not belong to the Bayesian network**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for P(targets|evs)

**Return type**

*pyAgrum.Potential* (page 53)

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what `makeInference` should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setEpsilon(eps)****Parameters****eps** (*float*) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$

**Return type**

None

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network

**setMaxIter(max)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `max <= 1`

**Return type**

None

**setMaxTime(timeout)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `timeout <= 0.0`

**Return type**

None

**setMinEpsilonRate(rate)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(p)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `p < 1`

**Return type**

None

**setTargets(targets)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity(v)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(evidces)**

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()****Returns**

True if the verbosity is enabled

**Return type**

bool

## Importance Sampling

**class pyAgrum.ImportanceSampling(bn)**

Class used for inferences using the Importance Sampling algorithm.

**ImportanceSampling(bn) -> ImportanceSampling****Parameters:**

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

**Parameters****bn** (*IBayesNet*) –**BN()****Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type***pyAgrum.IBayesNet***Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)****Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.



**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

• **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network

• [\*pyAgrum.UndefinedElement\*](#) (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a [\*pyAgrum.Potential\*](#) for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for  $P(\text{targets}|\text{evs})$

**Return type**

*pyAgrum.Potential* (page 53)

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**

*pyAgrum.OperationNotAllowed* (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()**

**Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setEpsilon(eps)****Parameters****eps** (*float*) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$ **Return type**

None

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network**setMaxIter(max)****Parameters****max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\max \leq 1$

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{timeout} \leq 0.0$

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity(*v*)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(*evidces*)**

Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If one value is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of a value is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If one value is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()**

**Returns**

True if the verbosity is enabled

**Return type**

bool

## Loopy sampling

### Loopy Gibbs Sampling

**class** `pyAgrum.LoopyGibbsSampling`(*bn*)

**Parameters**

**bn** (`IBayesNet`) –

**BN()**

**Returns**

A constant reference over the `IBayesNet` referenced by this class.

**Return type**

`pyAgrum.IBayesNet`

**Raises**

[`pyAgrum.UndefinedElement`](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node already has an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**burnIn()****Returns**

size of burn in on number of iteration

**Return type**

int

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**

[\*UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.**Returns**

a Potential for P(targets|evs)

**Return type***pyAgrum.Potential* (page 53)**hardEvidenceNodes()**



**Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**isDrawnAtRandom()****Returns**

True if variables are drawn at random

**Return type**

bool

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**makeInference\_()****Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrDrawnVar()****Returns**

the number of variable drawn at each iteration

**Return type**

int

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setBurnIn(*b*)****Parameters****b** (*int*) – size of burn in on number of iteration**Return type**

None

**setDrawnAtRandom(\_atRandom)****Parameters****\_atRandom** (*bool*) – indicates if variables should be drawn at random**Return type**

None

**setEpsilon(*eps*)****Parameters****eps** (*float*) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $eps < 0$

**Return type**

None

**setEvidence**(*evidces*)

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node pyAgrum.InvalidArgument If the size of a value is different from the domain side of the node pyAgrum.FatalError If one value is a vector of 0s pyAgrum.UndefinedElement If one node does not belong to the Bayesian network

**setMaxIter**(*max*)**Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If max <= 1

**Return type**

None

**setMaxTime**(*timeout*)**Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If timeout<=0.0

**Return type**

None

**setMinEpsilonRate**(*rate*)**Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setNbrDrawnVar**(*\_nbr*)**Parameters**

**\_nbr** (*int*) – the number of variables to be drawn at each iteration

**Return type**

None

**setPeriodSize**(*p*)**Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If p<1

**Return type**

None

**setTargets**(*targets*)

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity**(*v*)**Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**setVirtualLBPSize**(*vlbpsize*)**Parameters****vlbpsize** (*float*) – the size of the virtual LBP**Return type**

None

**softEvidenceNodes**()**Returns**

the set of nodes with soft evidence

**Return type**

set

**targets**()**Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence**(*evidces*)

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity**()**Returns**

True if the verbosity is enabled

**Return type**

bool

## Loopy Monte Carlo Sampling

**class** pyAgrum.LoopyMonteCarloSampling(*bn*)**Parameters****bn** (IBayesNet) –**BN**()**Returns**

A constant reference over the IBayesNet referenced by this class.

**Return type**

pyAgrum.IBayesNet

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shannon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

[\*pyAgrum.IndexError\*](#) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- [\*pyAgrum.IndexError\*](#) – If one of the node does not belong to the Bayesian network

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If node *Id* is not in the Bayesian network

**Return type**

None

**evidenceImpact**(*target, evs*)Create a `pyAgrum.Potential` for  $P(\text{target}|\text{evs})$  (for all instantiation of *target* and *evs*)**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some *evs* are d-separated, they are not included in the Potential.**Returns**a Potential for  $P(\text{targets}|\text{evs})$ **Return type**[\*pyAgrum.Potential\*](#) (page 53)**hardEvidenceNodes**()**Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence**(*\*args*)**Parameters**

- **id** (*int*) – a node *Id*
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasHardEvidence**(*nodeName*)**Parameters**

- **id** (*int*) – a node *Id*
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**hasSoftEvidence**(*\*args*)**Parameters**

- **id** (*int*) – a node *Id*
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**history**()**Returns**

the scheme history



**Return type**

tuple

**Raises**

[`pyAgrum.OperationNotAllowed`](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- [`pyAgrum.IndexError`](#) – If the node does not belong to the Bayesian network
- [`pyAgrum.UndefinedElement`](#) (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**makeInference\_()****Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setEpsilon(*eps*)****Parameters****eps** (*float*) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$ **Return type**

None

**setEvidence(*evidces*)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters****evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument`  
 If the size of a value is different from the domain side of the node `pyAgrum.FatalError`  
 If one value is a vector of 0s `pyAgrum.UndefinedElement`  
 If one node does not belong to the Bayesian network

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `max <= 1`

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `timeout <= 0.0`

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If `p < 1`

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity(*v*)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**setVirtualLBPSize(*vlbpsize*)****Parameters**

**vlbpsize** (*float*) – the size of the virtual LBP

**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()**

**Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(evidces)**

Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If one value is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of a value is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If one value is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()**

**Returns**

True if the verbosity is enabled

**Return type**

bool

## Loopy Weighted Sampling

**class** `pyAgrum.LoopyWeightedSampling(bn)`

**Parameters**

**bn** (`IBayesNet`) –

**BN()**

**Returns**

A constant reference over the `IBayesNet` referenced by this class.

**Return type**

`pyAgrum.IBayesNet`

**Raises**

- [`pyAgrum.UndefinedElement`](#) (page 291) – If no Bayes net has been assigned to the inference.

**H(\*args)**

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets()**

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type***pyAgrum.Potential* (page 53)**Raises***UndefinedElement* (page 291) – If an element of nodes is not in targets**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- *pyAgrum.UndefinedElement* (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**Create a *pyAgrum.Potential* for  $P(\text{target}|\text{evs})$  (for all instantiation of target and evs)**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for  $P(\text{targets}|\text{evs})$

**Return type**

*pyAgrum.Potential* (page 53)

**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()****Returns**

the scheme history

**Return type**

tuple

**Raises**

**pyAgrum.OperationNotAllowed** (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**makeInference\_()****Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()**



**Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**setEpsilon(eps)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$

**Return type**

None

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network

**setMaxIter(max)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\max \leq 1$

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{timeout} \leq 0.0$

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**setVerbosity(*v*)****Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**setVirtualLBPSize(*vlbpsize*)****Parameters**

**vlbpsize** (*float*) – the size of the virtual LBP

**Return type**

None

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence**(*evidces*)

Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If one value is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of a value is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If one value is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity**()**Returns**

True if the verbosity is enabled

**Return type**

bool

## Loopy Importance Sampling

**class** `pyAgrum.LoopyImportanceSampling`(*bn*)**Parameters**

**bn** (*IBayesNet*) –

**BN**()**Returns**

A constant reference over the *IBayesNet* referenced by this class.

**Return type**

`pyAgrum.IBayesNet`

**Raises**

- [`pyAgrum.UndefinedElement`](#) (page 291) – If no Bayes net has been assigned to the inference.

**H**(\**args*)**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**addAllTargets**()

Add all the nodes as targets.

**Return type**

None

**addEvidence**(\**args*)

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node already has an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of vals is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If vals is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**[`pyAgrum.UndefinedElement`](#) (page 291) – If target is not a NodeId in the Bayes net**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [`pyAgrum.InvalidArgument`](#) (page 288) – If the node does not already have an evidence
- [`pyAgrum.InvalidArgument`](#) (page 288) – If val is not a value for the node
- [`pyAgrum.InvalidArgument`](#) (page 288) – If the size of vals is different from the domain side of the node
- [`pyAgrum.FatalError`](#) (page 287) – If vals is a vector of 0s
- [`pyAgrum.UndefinedElement`](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**currentPosterior(\*args)**

Computes and returns the current posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the current posterior probability of the node

**Return type**[`pyAgrum.Potential`](#) (page 53)**Raises**[`UndefinedElement`](#) (page 291) – If an element of nodes is not in targets**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises****pyAgrum.IndexError** – If the node does not belong to the Bayesian network**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact(target, evs)**

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for P(targets|evs)

**Return type**[pyAgrum.Potential](#) (page 53)**hardEvidenceNodes()****Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence(\*args)**

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence(nodeName)**

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)**

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**history()**

**Returns**

the scheme history

**Return type**

tuple

**Raises**

**pyAgrum.OperationNotAllowed** (page 290) – If the scheme did not performed or if verbosity is set to false

**isTarget(\*args)**

**Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**makeInference()**

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later,

the computations of the posteriors can be done ‘lightly’ by multiplying and projecting those messages.

**Return type**

None

**makeInference\_()****Return type**

None

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrEvidence()****Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence()****Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**[\*pyAgrum.Potential\*](#) (page 53)**Raises**[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets**setEpsilon(*eps*)****Parameters****eps** (*float*) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$ **Return type**

None

**setEvidence(*evidces*)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters****evidces** (*dict*) – a dict of evidences**Raises**[\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node `pyAgrum.InvalidArgument` If the size of a value is different from the domain side of the node `pyAgrum.FatalError` If one value is a vector of 0s `pyAgrum.UndefinedElement` If one node does not belong to the Bayesian network**setMaxIter(*max*)****Parameters****max** (*int*) – the maximum number of iteration**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{max} \leq 1$ **Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{timeout} \leq 0.0$ **Return type**

None



**setMinEpsilonRate**(*rate*)

**Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize**(*p*)

**Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If  $p < 1$

**Return type**

None

**setTargets**(*targets*)

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[pyAgrum.UndefinedElement](#) (page 291) – If one target is not in the Bayes net

**setVerbosity**(*v*)

**Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**setVirtualLBPSize**(*vlbpsize*)

**Parameters**

**vlbpsize** (*float*) – the size of the virtual LBP

**Return type**

None

**softEvidenceNodes**()

**Returns**

the set of nodes with soft evidence

**Return type**

set

**targets**()

**Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence**(*evidces*)

Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [pyAgrum.InvalidArgument](#) (page 288) – If one value is not a value for the node
- [pyAgrum.InvalidArgument](#) (page 288) – If the size of a value is different from the domain side of the node
- [pyAgrum.FatalError](#) (page 287) – If one value is a vector of 0s
- [pyAgrum.UndefinedElement](#) (page 291) – If one node does not belong to the Bayesian network

**verbosity()**

**Returns**

True if the verbosity is enabled

**Return type**

bool

## 1.4.6 Learning

pyAgrum encloses all the learning processes for Bayesian network in a simple class `BNLearner`. This class gives access directly to the complete learning algorithm and theirs parameters (such as prior, scores, constraints, etc.) but also proposes low-level functions that eases the work on developing new learning algorithms (for instance, compute chi2 or conditionnal likelihood on the database, etc.).

**class** `pyAgrum.BNLearner`(*filename*, *inducedTypes=True*)

**Parameters:**

- **source** (*str* or *pandas.DataFrame*) – the data to learn from
- **missingSymbols** (*List[str]*) – list of string that will be interpreted as missing values (by default : ['?'])
- **inducedTypes** (*Bool*) – whether `BNLearner` should try to automatically find the type of each variable

**BNLearner**(*filename*,*src*) -> **BNLearner**

**Parameters:**

- **source** (*str* or *\*pandas.DataFrame*) – the data to learn from
- **src** (*pyAgrum.BayesNet*) – the Bayesian network used to find those modalities
- **missingSymbols** (*List[str]*) – list of string that will be interpreted as missing values (by default : ['?'])

**BNLearner**(*learner*) -> **BNLearner**

**Parameters:**

- **learner** (*pyAgrum.BNLearner*) – the `BNLearner` to copy

**G2**(\**args*)

G2 computes the G2 statistic and pvalue for two columns, given a list of other columns.

**Parameters**

- **name1** (*str*) – the name of the first column
- **name2** (*str*) – the name of the second column
- **knowing** (*[str]*) – the list of names of conditioning columns

**Returns**

the G2 statistic and the associated p-value as a Tuple

**Return type**

statistic,pvalue

**addForbiddenArc**(\**args*)

**Return type**

[\*BNLearner\*](#) (page 182)

**addMandatoryArc**(\**args*)

**Return type**

[\*BNLearner\*](#) (page 182)

**addPossibleEdge**(\**args*)

**Return type**

[\*BNLearner\*](#) (page 182)

**chi2(\*args)**

chi2 computes the chi2 statistic and pvalue for two columns, given a list of other columns.

**Parameters**

- **name1** (*str*) – the name of the first column
- **name2** (*str*) – the name of the second column
- **knowing** (*[str]*) – the list of names of conditioning columns

**Returns**

the chi2 statistic and the associated p-value as a Tuple

**Return type**

statistic,pvalue

**correctedMutualInformation(\*args)****Return type**

float

**currentTime()****Returns**

get the current running time in second (float)

**Return type**

float

**databaseWeight()****Return type**

float

**domainSize(\*args)****Return type**

int

**epsilon()****Returns**

the value of epsilon

**Return type**

float

**eraseForbiddenArc(\*args)****Return type**

[BNLearner](#) (page 182)

**eraseMandatoryArc(\*args)****Return type**

[BNLearner](#) (page 182)

**erasePossibleEdge(\*args)****Return type**

[BNLearner](#) (page 182)

**fitParameters(bn)**

Easy shortcut to LearnParameters method. fitParameters uses self to directly populate the CPTs of bn.0

**Parameters**

**bn** ([pyAgrum.BayesNet](#) (page 64)) – a BN which will directly have its parameters learned.

**getNumberOfThreads()****Return type**

int

**hasMissingValues()****Return type**

bool

**history()**

**Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**idFromName(*var\_name*)**

**Parameters**

**var\_name** (str) –

**Return type**

int

**isGumNumberOfThreadsOverriden()**

**Return type**

bool

**latentVariables()**

<b>Warning:</b> learner must be using 3off2 or MIIC algorithm
---

**Returns**

the list of latent variables

**Return type**

list

**learnBN()**

learn a BayesNet from a file (must have read the db before)

**Returns**

the learned BayesNet

**Return type**

[\*pyAgrum.BayesNet\*](#) (page 64)

**learnDAG()**

**Return type**

[\*DAG\*](#) (page 8)

**learnEssentialGraph()**

**learnMixedStructure()**

**Return type**

[\*MixedGraph\*](#) (page 20)

**learnParameters(\*args)**

learns a BN (its parameters) when its structure is known.

**Parameters**

- **dag** ([\*pyAgrum.DAG\*](#) (page 8)) –
- **bn** ([\*pyAgrum.BayesNet\*](#) (page 64)) –
- **take\_into\_account\_score** (bool) – The dag passed in argument may have been learnt from a structure learning. In this case, if the score used to learn the structure has an implicit prior (like K2 which has a 1-smoothing prior), it is important to also take into account this implicit prior for parameter learning. By default, if a score exists, we will learn parameters by taking into account the prior specified by methods `usePriorXXX ()` + the implicit prior of the score, else we just take into account the prior specified by `usePriorXXX ()`

**Returns**

the learned BayesNet

**Return type***pyAgrum.BayesNet* (page 64)**Raises**

- **pyAgrum.MissingVariableInDatabase** – If a variable of the BN is not found in the database
- **pyAgrum.UnknownLabelInDatabase** (page 291) – If a label is found in the database that do not correspond to the variable

**logLikelihood(\*args)****Return type**

float

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**mutualInformation(\*args)****Return type**

float

**nameFromId(id)****Parameters****id** (int) –**Return type**

str

**names()****Return type**

List[str]

**nbCols()****Return type**

int

**nbRows()****Return type**

int

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$ **pseudoCount(*vars*)**

access to pseudo-count (priors taken into account)

**Parameters****vars** (*list[str]*) – a list of name of vars to add in the pseudo\_count**Return type**

a Potential containing this pseudo-counts

**rawPseudoCount(\*args)****Return type**

List[float]

**recordWeight(*i*)****Parameters****i** (int) –**Return type**

float

**score(\*args)****Return type**

float

**setDatabaseWeight(*new\_weight*)****Parameters****new\_weight** (float) –**Return type**

None

**setEpsilon(*eps*)****Parameters****eps** (float) – the epsilon we want to use**Raises**[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$ **Return type**

None

**setForbiddenArcs(*set*)****Parameters****set** (Set[Tuple[int, int]]) –**Return type**[\*BNLearner\*](#) (page 182)**setInitialDAG(*dag*)****Parameters****dag** ([\*pyAgrum.DAG\*](#) (page 8)) – an initial DAG structure**Return type**[\*BNLearner\*](#) (page 182)**setMandatoryArcs(*set*)****Parameters****set** (Set[Tuple[int, int]]) –

**Return type**[BNLearner](#) (page 182)**setMaxIndegree**(*max\_indegree*)**Parameters****max\_indegree** (*int*) – the limit number of parents**Return type**[BNLearner](#) (page 182)**setMaxIter**(*max*)**Parameters****max** (*int*) – the maximum number of iteration**Raises**[pyAgrum.OutOfBounds](#) (page 290) – If max <= 1**Return type**

None

**setMaxTime**(*timeout*)**Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**[pyAgrum.OutOfBounds](#) (page 290) – If timeout<=0.0**Return type**

None

**setMinEpsilonRate**(*rate*)**Parameters****rate** (*float*) – the minimal epsilon rate**Return type**

None

**setNumberOfThreads**(*nb*)**Parameters****nb** (*int*) –**Return type**

None

**setPeriodSize**(*p*)**Parameters****p** (*int*) – number of samples between 2 stopping**Raises**[pyAgrum.OutOfBounds](#) (page 290) – If p<1**Return type**

None

**setPossibleEdges**(*set*)**Parameters****set** (Set[Tuple[int, int]]) –**Return type**[BNLearner](#) (page 182)**setPossibleSkeleton**(*skeleton*)**Parameters****skeleton** ([UndiGraph](#) (page 12)) –**Return type**[BNLearner](#) (page 182)**setRecordWeight**(*i*, *weight*)**Parameters**

- **i** (*int*) –

- **weight** (float) –

**Return type**

None

**setSliceOrder**(\*args)

**Return type**

[BNLearner](#) (page 182)

**setVerbosity**(v)

**Parameters**

**v** (bool) – verbosity

**Return type**

None

**state**()

**Return type**

object

**use3off2**()

Indicate that we wish to use 3off2.

**Return type**

[BNLearner](#) (page 182)

**useAprioriBDeu**()

Deprecated methods in BNLearner for pyAgrum>1.1.1

**useAprioriDirichlet**()

Deprecated methods in BNLearner for pyAgrum>1.1.1

**useAprioriSmoothing**()

Deprecated methods in BNLearner for pyAgrum>1.1.1

**useBDeuPrior**(weight=1.0)

**Parameters**

**weight** (float) –

**Return type**

[BNLearner](#) (page 182)

**useDirichletPrior**(\*args)

Use the Dirichlet prior.

**Parameters**

- **source** (str/[pyAgrum.BayesNet](#) (page 64)) – the Dirichlet related source (filename of a database or a Bayesian network)
- **weight** (float (optional)) – the weight of the prior (the ‘size’ of the corresponding ‘virtual database’)

**Return type**

[BNLearner](#) (page 182)

**useEM**(epsilon)

Indicates if we use EM for parameter learning.

**Parameters**

**epsilon** (float) – if epsilon=0.0 then EM is not used if epsilon>0 then EM is used and stops when the sum of the cumulative squared error on parameters is less than epsilon.

**Return type**

[BNLearner](#) (page 182)

**useGreedyHillClimbing**()

Indicate that we wish to use a greedy hill climbing algorithm.

**Return type**

[BNLearner](#) (page 182)



**useK2**(\*args)

**Return type**

[BNLearner](#) (page 182)

**useLocalSearchWithTabuList**(*tabu\_size=100, nb\_decrease=2*)

**Parameters**

- **tabu\_size** (int) –
- **nb\_decrease** (int) –

**Return type**

[BNLearner](#) (page 182)

**useMDLCorrection**()

Indicate that we wish to use the MDL correction for 3off2 or MIIC

**Return type**

[BNLearner](#) (page 182)

**useMIIC**()

Indicate that we wish to use MIIC.

**Return type**

[BNLearner](#) (page 182)

**useNMLCorrection**()

Indicate that we wish to use the NML correction for 3off2 or MIIC

**Return type**

[BNLearner](#) (page 182)

**useNoApriori**()

Deprecated methods in BNLearner for pyAgrum>1.1.1

**useNoCorrection**()

Indicate that we wish to use the NoCorr correction for 3off2 or MIIC

**Return type**

[BNLearner](#) (page 182)

**useNoPrior**()

Use no prior.

**Return type**

[BNLearner](#) (page 182)

**useScoreAIC**()

Indicate that we wish to use an AIC score.

**Return type**

[BNLearner](#) (page 182)

**useScoreBD**()

Indicate that we wish to use a BD score.

**Return type**

[BNLearner](#) (page 182)

**useScoreBDeu**()

Indicate that we wish to use a BDeu score.

**Return type**

[BNLearner](#) (page 182)

**useScoreBIC**()

Indicate that we wish to use a BIC score.

**Return type**

[BNLearner](#) (page 182)

**useScoreK2()**

Indicate that we wish to use a K2 score.

**Return type**

[BNLearner](#) (page 182)

**useScoreLog2Likelihood()**

Indicate that we wish to use a Log2Likelihood score.

**Return type**

[BNLearner](#) (page 182)

**useSmoothingPrior(*weight=1*)**

Use the prior smoothing.

**Parameters**

**weight** (*float*) – pass in argument a weight if you wish to assign a weight to the smoothing, otherwise the current weight of the learner will be used.

**Return type**

[BNLearner](#) (page 182)

**verbosity()****Returns**

True if the verbosity is enabled

**Return type**

bool

## 1.5 Influence Diagram

An influence diagram is a compact graphical and mathematical representation of a decision situation. It is a generalization of a Bayesian network, in which not only probabilistic inference problems but also decision making problems (following the maximum expected utility criterion) can be modeled and solved. It includes 3 types of nodes : action, decision and utility nodes ([from wikipedia \(https://en.wikipedia.org/wiki/Influence\\_diagram\)](https://en.wikipedia.org/wiki/Influence_diagram)).

PyAgrum's so-called influence diagram represents both influence diagrams and LIMIDs. The way to enforce that such a model represent an influence diagram and not a LIMID belongs to the inference engine.

**Tutorial**

- [Tutorial on Influence Diagram](#) (page 363)

**Reference**

### 1.5.1 Model for Decision in PGM

```
class pyAgrum.InfluenceDiagram(*args)
```

InfluenceDiagram represents an Influence Diagram.

**InfluenceDiagram() -> InfluenceDiagram**

default constructor

**InfluenceDiagram(source) -> InfluenceDiagram**

**Parameters:**

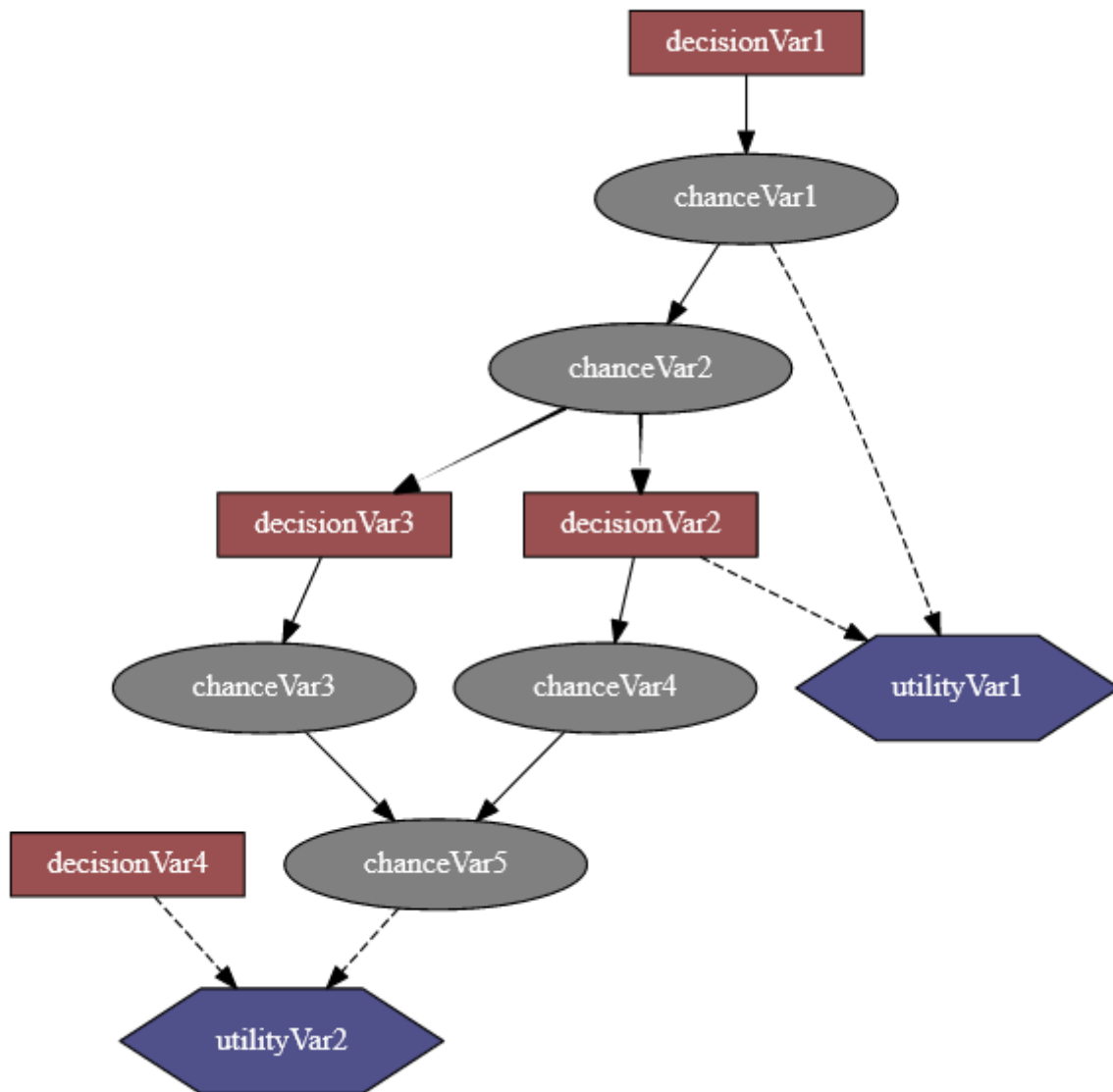
- **source** (*pyAgrum.InfluenceDiagram*) – the InfluenceDiagram to copy

```
add(*args)
```

Add a variable, it's associate node and it's CPT.

The id of the new variable is automatically generated.

**Parameters**



- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – The variable added by copy that will be a chance node.
- **descr** (*str*) – the descr of the variable following *fast syntax* (page 279) extended for [pyAgrum.fastID\(\)](#) (page 281).
- **nbr\_mod\_or\_id** (*int*) – if the first argument is *variable*, this set an optional fixed id for the node. If the first argument is *descr*, this gives the default number of modalities for the variable. Note that if a utility node is described in *descr*, this value is overridden by 1.

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[pyAgrum.DuplicateElement](#) (page 287) – If already used id or name.

**addArc(\*args)**

Add an arc in the ID, and update diagram's potential nodes cpt if necessary.

**Parameters**

- **tail** (*Union[int, str]*) – a variable's id (int) or name
- **head** (*Union[int, str]*) – a variable's id (int) or name

**Raises**

- [pyAgrum.InvalidEdge](#) (page 289) – If arc.tail and/or arc.head are not in the ID.
- [pyAgrum.InvalidEdge](#) (page 289) – If tail is a utility node

**Return type**

None

**addArcs(listArcs)**

add a list of arcs in te model.

**Parameters**

**listArcs** (*List[Tuple[int, int]]*) – the list of arcs

**addChanceNode(\*args)**

Add a chance variable, it's associate node and it's CPT.

The id of the new variable is automatically generated.

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – the variable added by copy.
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose.

**Warning:** give an id (not 0) should be reserved for rare and specific situations !!!

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[pyAgrum.DuplicateElement](#) (page 287) – If id(<>0) is already used

**addDecisionNode(\*args)**

Add a decision variable.

The id of the new variable is automatically generated.

**Parameters**

- **variable** ([pyAgrum.DiscreteVariable](#) (page 25)) – the variable added by copy.
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose.

**Warning:** give an id (not 0) should be reserved for rare and specific situations !!!

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

[\*pyAgrum.DuplicateElement\*](#) (page 287) – If id(<>0) is already used

**addStructureListener**(*whenNodeAdded=None, whenNodeDeleted=None, whenArcAdded=None, whenArcDeleted=None*)

Add the listeners in parameters to the list of existing ones.

**Parameters**

- **whenNodeAdded** (*lambda expression*) – a function for when a node is added
- **whenNodeDeleted** (*lambda expression*) – a function for when a node is removed
- **whenArcAdded** (*lambda expression*) – a function for when an arc is added
- **whenArcDeleted** (*lambda expression*) – a function for when an arc is removed

**addUtilityNode**(\*args)

Add a utility variable, it's associate node and it's UT.

The id of the new variable is automatically generated.

**Parameters**

- **variable** ([\*pyAgrum.DiscreteVariable\*](#) (page 25)) – the variable added by copy
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose

**Warning:** give an id (not 0) should be reserved for rare and specific situations !!!

**Returns**

the id of the added variable.

**Return type**

int

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If variable has more than one label
- [\*pyAgrum.DuplicateElement\*](#) (page 287) – If id(<>0) is already used

**addVariables**(*listFastVariables, default\_nbr\_mod=2*)

Add a list of variable in the form of 'fast' syntax.

**Parameters**

- **listFastVariables** (*List[str]*) – the list of variables following *fast syntax* (page 279) extended for [\*pyAgrum.fastID\(\)\*](#) (page 281).
- **default\_nbr\_mod** (*int*) – the number of modalities for the variable if not specified in the fast description. Note that default\_nbr\_mod=1 is mandatory to create variables with only one modality (for utility for instance).

**Returns**

the list of created ids.

**Return type**

List[int]

**ancestors**(*norid*)

**Parameters**

**norid** (object) –

**Return type**

object

**arcs()**

**Returns**

the list of all the arcs in the Influence Diagram.

**Return type**

list

**chanceNodeSize()**

**Returns**

the number of chance nodes.

**Return type**

int

**changeVariableName(\*args)**

**Parameters**

- **var** (*Union[int, str]*) – a variable's id (int) or name
- **new\_name** (*str*) – the name of the variable

**Raises**

- [\*pyAgrum.DuplicateLabel\*](#) (page 287) – If this name already exists
- [\*pyAgrum.NotFound\*](#) (page 289) – If no nodes matches id.

**Return type**

None

**children(*norid*)**

**Parameters**

- **var** (*Union[int, str]*) – a variable's id (int) or name
- **norid** (object) –

**Returns**

the set of all the children

**Return type**

Set

**clear()**

**Return type**

None

**completeInstantiation()**

**Return type**

[\*Instantiation\*](#) (page 47)

**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**cpt(\*args)**

Returns the CPT of a variable.

**Parameters**

**var** (*Union[int, str]*) – a variable's id (int) or name

**Returns**

The variable's CPT.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[`pyAgrum.NotFound`](#) (page 289) – If no variable’s id matches varId.

**dag()****Returns**

a constant reference to the dag of this BayesNet.

**Return type**

[`pyAgrum.DAG`](#) (page 8)

**decisionNodeSize()****Returns**

the number of decision nodes

**Return type**

int

**decisionOrder()****decisionOrderExists()****Returns**

True if a directed path exist with all decision node

**Return type**

bool

**descendants(*norid*)****Parameters**

**norid** (object) –

**Return type**

object

**empty()****Return type**

bool

**erase(\*args)**

Erase a Variable from the network and remove the variable from all his childs.

If no variable matches the id, then nothing is done.

**Parameters**

- **id** (*int*) – The id of the variable to erase.
- **var** (*Union[int, str, pyAgrum.DiscreteVariable* (page 25)] – a variable’s id (int) or name or th reference on the variable to remove.

**Return type**

None

**eraseArc(\*args)**

Removes an arc in the ID, and update diagram’s potential nodes cpt if necessary.

If (tail, head) doesn’t exist, the nothing happens.

**Parameters**

- **arc** ([`pyAgrum.Arc`](#) (page 3)) – The arc to be removed whn calling `eraseArc(arc)`
- **tail** (*Union[int, str]*) – a variable’s id (int) or name when calling `eraseArc(tail,head)`
- **head** (*Union[int, str]*) – a variable’s id (int) or name when calling `eraseArc(tail,head)`

**Return type**

None

**exists(*node*)****Parameters**

**node** (int) –

**Return type**

bool

**existsArc**(\*args)**Return type**

bool

**existsPathBetween**(\*args)**Returns**

true if a path exists between two nodes.

**Return type**

bool

**family**(*norid*)**Parameters****norid** (object) –**Return type**

object

**static fastPrototype**(*dotlike*, *domainSize=2*)**Create an Influence Diagram with a dot-like syntax which specifies:**

- the structure ‘a->b<-c;b->d;c<-e;’.
- a prefix for the type of node (chance/decision/utility nodes):
  - *a* : a chance node named ‘a’ (by default)
  - *\$a* : a utility node named ‘a’
  - *\*a* : a decision node named ‘a’
- the type of the variables with different syntax as postfix:
  - by default, a variable is a pyAgrum.RangeVariable using the default domain size (second argument)
  - with ‘*a[10]*’, the variable is a pyAgrum.RangeVariable using 10 as domain size (from 0 to 9)
  - with ‘*a[3,7]*’, the variable is a pyAgrum.RangeVariable using a domainSize from 3 to 7
  - with ‘*a[1,3.14,5,6.2]*’, the variable is a pyAgrum.DiscretizedVariable using the given ticks (at least 3 values)
  - with ‘*a{top|middle|bottom}*’, the variable is a pyAgrum.LabelizedVariable using the given labels.
  - with ‘*a{-1|5|0|3}*’, the variable is a pyAgrum.IntegerVariable using the sorted given values.
  - with ‘*a{-0.5|5.01|0|3.1415}*’, the variable is a pyAgrum.NumericalDiscreteVariable using the sorted given values.

---

**Note:**

- If the dot-like string contains such a specification more than once for a variable, the first specification will be used.
  - the potentials (probabilities, utilities) are randomly generated.
  - see also pyAgrum.fastID.
-



## Examples

```
>>> import pyAgrum as gum
>>> bn=pyAgrum.fastID('A->B[1,3]<-*C{yes|No}->$D<-E[1,2.5,3.9]',6)
```

### Parameters

- **dotlike** (*str*) – the string containing the specification
- **domainSize** (*int*) – the default domain size for variables

### Returns

the resulting Influence Diagram

### Return type

*pyAgrum.InfluenceDiagram* (page 190)

### getDecisionGraph()

#### Returns

the temporal Graph.

#### Return type

*pyAgrum.DAG* (page 8)

### hasSameStructure(*other*)

#### Parameters

**pyAgrum.DAGmodel** – a direct acyclic model

#### Returns

True if all the named node are the same and all the named arcs are the same

#### Return type

bool

### idFromName(*name*)

Returns a variable's id given its name.

#### Parameters

**name** (*str*) – the variable's name from which the id is returned.

#### Returns

the variable's node id.

#### Return type

int

#### Raises

*pyAgrum.NotFound* (page 289) – If no such name exists in the graph.

### ids(*names*)

### isChanceNode(\**args*)

#### Parameters

**varId** (*int*) – the tested node id.

#### Returns

true if node is a chance node

#### Return type

bool

### isDecisionNode(\**args*)

#### Parameters

**varId** (*int*) – the tested node id.

#### Returns

true if node is a decision node

#### Return type

bool

### isIndependent(\**args*)

#### Return type

bool

**isUtilityNode(\*args)**

**Parameters**

**varId** (*int*) – the tested node id.

**Returns**

true if node is an utility node

**Return type**

bool

**loadBIFXML(\*args)**

Load a BIFXML file.

**Parameters**

**name** (*str*) – the name's file

**Raises**

- [pyAgrum.IOError](#) (page 288) – If file not found
- [pyAgrum.FatalError](#) (page 287) – If file is not valid

**Return type**

bool

**log10DomainSize()**

**Return type**

float

**moralGraph(clear=True)**

Returns the moral graph of the BayesNet, formed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected.

**Returns**

The moral graph

**Return type**

[pyAgrum.UndiGraph](#) (page 12)

**Parameters**

**clear** (bool) –

**moralizedAncestralGraph(nodes)**

**Parameters**

**nodes** (object) –

**Return type**

[UndiGraph](#) (page 12)

**names()**

**Returns**

The names of the InfluenceDiagram variables

**Return type**

List[str]

**nodeId(var)**

**Parameters**

**var** ([pyAgrum.DiscreteVariable](#) (page 25)) – a variable

**Returns**

the id of the variable

**Return type**

int

**Raises**

[pyAgrum.IndexError](#) – If the InfluenceDiagram does not contain the variable

**nodes()**

**Returns**

the set of ids

**Return type**

set

**nodeset**(*names*)

**Parameters**

**names** (Vector\_string) –

**Return type**

List[int]

**parents**(*norid*)

**Parameters**

- **var** (*Union[int, str]*) – a variable's id (int) or name
- **norid** (object) –

**Returns**

the set of the parents ids.

**Return type**

set

**saveBIFXML**(*name*)

Save the BayesNet in a BIFXML file.

**Parameters**

**name** (*str*) – the file's name

**Return type**

None

**size**()

**Returns**

the number of nodes in the graph

**Return type**

int

**sizeArcs**()

**Returns**

the number of arcs in the graph

**Return type**

int

**property thisown**

The membership flag

**toDot**()

**Returns**

a friendly display of the graph in DOT format

**Return type**

str

**topologicalOrder**(*clear=True*)

**Returns**

the list of the nodes Ids in a topological order

**Return type**

List

**Raises**

[\*pyAgrum.InvalidDirectedCycle\*](#) (page 288) – If this graph contains cycles

**Parameters**

**clear** (bool) –

**utility**(\**args*)

**Parameters**

**var** (*Union[int, str]*) – a variable's id (int) or name

**Returns**

the utility table of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

Raises

**pyAgrum.IndexError** – If the InfluenceDiagram does not contain the variable

**utilityNodeSize()**

Returns

the number of utility nodes

**Return type**

int

**variable(\*args)**

Parameters

**id** (*int*) – the node id

Returns

a constant reference over a variable given it's node id

**Return type**

[pyAgrum.DiscreteVariable](#) (page 25)

Raises

**pyAgrum.NotFound** (page 289) – If no variable's id matches the parameter

**variableFromName(name)**

Parameters

**name** (*str*) – a variable's name

Returns

the variable

**Return type**

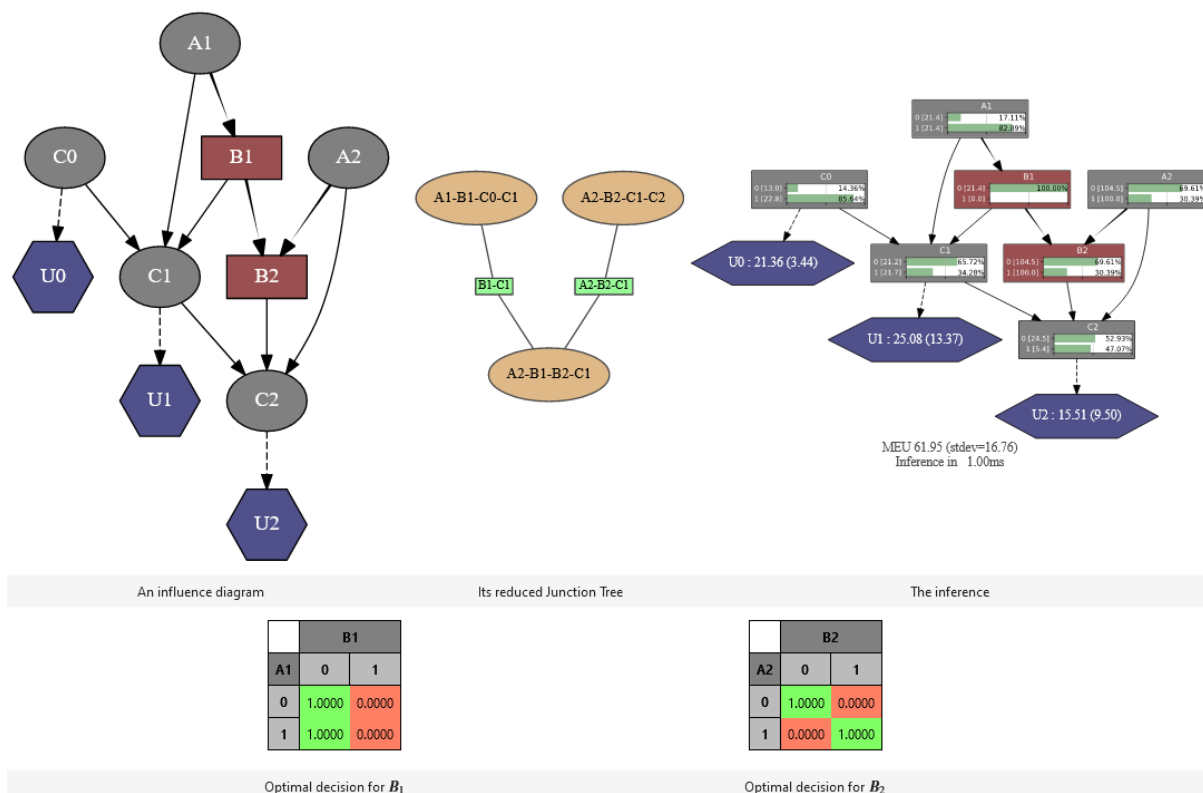
[pyAgrum.DiscreteVariable](#) (page 25)

Raises

**pyAgrum.IndexError** – If the InfluenceDiagram does not contain the variable

**variableNodeMap()**

## 1.5.2 Inference for Influence Diagram



**class** pyAgrum.**ShaferShenoyLIMIDInference**(*infDiag*)

This inference considers the provided model as a LIMID rather than an influence diagram. It is an optimized implementation of the LIMID resolution algorithm. However an inference on a classical influence diagram can be performed by adding a assumption of the existence of the sequence of decision nodes to be solved, which also implies that the decision choices can have an impact on the rest of the sequence (Non Forgetting Assumption, cf. pyAgrum.ShaferShenoyLIMIDInference.addNoForgettingAssumption).

**Parameters**

**infDiag** (*InfluenceDiagram* (page 190)) –

**MEU**(\*args)

Returns maximum expected utility obtained from inference.

**Raises**

*pyAgrum.OperationNotAllowed* (page 290) – If no inference have yet been made

**Return type**

object

**addEvidence**(\*args)

**Return type**

None

**addNoForgettingAssumption**(\*args)

**Return type**

None

**chgEvidence**(\*args)

**Return type**

None

**clear**()

**Return type**

None

**eraseAllEvidence**()

Removes all the evidence entered into the diagram.

**Return type**

None

**eraseEvidence**(\*args)

**Parameters**

**evidence** (*pyAgrum.Potential* (page 53)) – the evidence to remove

**Raises**

*pyAgrum.IndexError* – If the evidence does not belong to the influence diagram

**Return type**

None

**hardEvidenceNodes**()

**Return type**

object

**hasEvidence**(\*args)

**Return type**

bool

**hasHardEvidence**(*nodeName*)

**Parameters**

**nodeName** (str) –

**Return type**

bool

**hasNoForgettingAssumption()****Return type**

bool

**hasSoftEvidence(\*args)****Return type**

bool

**influenceDiagram()**

Returns a constant reference over the InfluenceDiagram on which this class work.

**Returns**

the InfluenceDiagram on which this class work

**Return type**

[\*pyAgrum.InfluenceDiagram\*](#) (page 190)

**isSolvable()****Return type**

bool

**junctionTree()****makeInference()**

Makes the inference.

**Return type**

None

**meanVar(\*args)****Return type**

object

**nbrEvidence()****Return type**

int

**nbrHardEvidence()****Return type**

int

**nbrSoftEvidence()****Return type**

int

**optimalDecision(\*args)**

Returns best choice for decision variable given in parameter ( based upon MEU criteria )

**Parameters**

**decisionId** (*int*, *str*) – the id or name of the decision variable

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If no inference have yet been made

**pyAgrum.InvalidNode**

If node given in parmaeter is not a decision node

**Return type**

[\*Potential\*](#) (page 53)

**posterior(\*args)****Return type**

[\*Potential\*](#) (page 53)

**posteriorUtility(\*args)**

**Return type**

*Potential* (page 53)

**reducedGraph()**

**Return type**

*DAG* (page 8)

**reducedLIMID()**

**Return type**

*InfluenceDiagram* (page 190)

**reversePartialOrder()**

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- *pyAgrum.InvalidArgument* (page 288) – If one value is not a value for the node
- *pyAgrum.InvalidArgument* (page 288) – If the size of a value is different from the domain side of the node
- *pyAgrum.FatalError* (page 287) – If one value is a vector of 0s
- *pyAgrum.UndefinedElement* (page 291) – If one node does not belong to the influence diagram

**softEvidenceNodes()**

**Return type**

object

**updateEvidence(evidces)**

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- *pyAgrum.InvalidArgument* (page 288) – If one value is not a value for the node
- *pyAgrum.InvalidArgument* (page 288) – If the size of a value is different from the domain side of the node
- *pyAgrum.FatalError* (page 287) – If one value is a vector of 0s
- *pyAgrum.UndefinedElement* (page 291) – If one node does not belong to the Bayesian network

## 1.6 Credal Network

Credal networks are probabilistic graphical models based on imprecise probability. Credal networks can be regarded as an extension of Bayesian networks, where credal sets replace probability mass functions in the specification of the local models for the network variables given their parents. As a Bayesian network defines a joint probability mass function over its variables, a credal network defines a joint credal set (from Wikipedia ([https://en.wikipedia.org/wiki/Credal\\_network](https://en.wikipedia.org/wiki/Credal_network))).

### Tutorial

- *Tutorial on Credal Networks* (page 375)

### Reference

### 1.6.1 CN Model

**class** pyAgrum.CredalNet(\*args)

Constructor used to create a CredalNet (step by step or with two BayesNet)

**CredalNet()** -> CredalNet

default constructor

CredalNet(src\_min\_num,src\_max\_den) -> CredalNet

**Parameters**

- **src\_min\_num** (*str* or [pyAgrum.BayesNet](#) (page 64)) – The path to a BayesNet or the BN itself which contains lower probabilities.
- **src\_max\_den** (*str* or [pyAgrum.BayesNet](#) (page 64)) – The (optional) path to a BayesNet or the BN itself which contains upper probabilities.

**NodeType\_Credal** = 1

**NodeType\_Indic** = 3

**NodeType\_Precise** = 0

**NodeType\_Vacuous** = 2

**addArc**(tail, head)

Adds an arc between two nodes

**Parameters**

- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

**Raises**

- **pyAgrum.InvalidDirectedCircle** – If any (directed) cycle is created by this arc
- **pyAgrum.InvalidNode** (page 289) – If head or tail does not belong to the graph nodes
- **pyAgrum.DuplicateElement** (page 287) – If one of the arc already exists

**Return type**

None

**addVariable**(name, card)

**Parameters**

- **name** (*str*) – the name of the new variable
- **card** (*int*) – the domainSize of the new variable

**Returns**

the id of the new node

**Return type**

int

**approximatedBinarization**()

Approximate binarization.

Each bit has a lower and upper probability which is the lowest - resp. highest - over all vertices of the credal set. Enlarge the original credal sets and may induce huge imprecision.

**Warning:** Enlarge the original credal sets and therefor induce huge imprecision by propagation. Not recommended, use MCSampling or something else instead

**Return type**

None



**bnToCredal**(*beta*, *oneNet*, *keepZeroes=False*)

Perturbates the BayesNet provided as input for this CredalNet by generating intervals instead of point probabilities and then computes each vertex of each credal set.

**Parameters**

- **beta** (*float*) – The beta used to perturbate the network
- **oneNet** (*bool*) – used as a flag. Set to True if one BayesNet is provided with counts, to False if two BayesNets are provided; one with probabilities (the lower net) and one with denominators over the first modalities (the upper net)
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept

**Return type**

None

**computeBinaryCPTMinMax**()

**Return type**

None

**credalNet\_currentCpt**()

**Warning:** Experimental function - Return type to be wrapped

**Returns**

a constant reference to the (up-to-date) CredalNet CPTs.

**Return type**

tbw

**credalNet\_srcCpt**()

**Warning:** Experimental function - Return type to be wrapped

**Returns**

a constant reference to the (up-to-date) CredalNet CPTs.

**Return type**

tbw

**currentNodeType**(*id*)

**Parameters**

**id** (*int*) – The constant reference to the chosen NodeId

**Returns**

the type of the chosen node in the (up-to-date) CredalNet `__current_bn` if any, `__src_bn` otherwise.

**Return type**

[\*pyAgrum.CredalNet\*](#) (page 204)

**current\_bn**()

**Returns**

Returns a constant reference to the actual BayesNet (used as a DAG, its CPTs does not matter).

**Return type**

[\*pyAgrum.BayesNet\*](#) (page 64)

**domainSize**(*id*)

**Parameters**

**id** (*int*) – The id of the node

**Returns**

The cardinality of the node

**Return type**

int

**epsilonMax()****Returns**

a constant reference to the highest perturbation of the BayesNet provided as input for this CredalNet.

**Return type**

float

**epsilonMean()****Returns**

a constant reference to the average perturbation of the BayesNet provided as input for this CredalNet.

**Return type**

float

**epsilonMin()****Returns**

a constant reference to the lowest perturbation of the BayesNet provided as input for this CredalNet.

**Return type**

float

**fillConstraint(\*args)**

Set the interval constraints of a credal set of a given node (from an instantiation index)

**Parameters**

- **id** (*int*) – The id of the node
- **entry** (*int*) – The index of the instantiation excluding the given node (only the parents are used to compute the index of the credal set)
- **ins** ([pyAgrum.Instantiation](#) (page 47)) – The Instantiation
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

**Warning:** You need to call `intervalToCredal` when done filling all constraints.

**Warning:** DOES change the BayesNet (s) associated to this credal net !

**Return type**

None

**fillConstraints(id, lower, upper)**

Set the interval constraints of the credal sets of a given node (all instantiations)

**Parameters**

- **id** (*int*) – The id of the node
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

**Warning:** You need to call `intervalToCredal` when done filling all constraints.

**Warning:** DOES change the BayesNet (s) associated to this credal net !

**Return type**

None

**get\_binaryCPT\_max()**

**Warning:** Experimental function - Return type to be wrapped

**Returns**

a constant reference to the upper probabilities of each node X over the 'True' modality

**Return type**

tbw

**get\_binaryCPT\_min()**

**Warning:** Experimental function - Return type to be wrapped

**Returns**

a constant reference to the lower probabilities of each node X over the 'True' modality

**Return type**

tbw

**hasComputedBinaryCPTMinMax()**

**Return type**

bool

**idmLearning(*s=0, keepZeroes=False*)**

Learns parameters from a BayesNet storing counts of events.

Use this method when using a single BayesNet storing counts of events. IDM model if  $s > 0$ , standard point probability if  $s = 0$  (default value if none precised).

**Parameters**

- **s** (*int*) – the IDM parameter.
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept.

**Return type**

None

**instantiation(*id*)**

Get an Instantiation from a node id, usefull to fill the constraints of the network.

bnet accessors / shortcuts.

**Parameters**

**id** (*int*) – the id of the node we want an instantiation from

**Returns**

the instantiation

**Return type**

[\*pyAgrum.Instantiation\*](#) (page 47)

**intervalToCredal()**

Computes the vertices of each credal set according to their interval definition (uses lrs).

Use this method when using two BayesNet, one with lower probabilities and one with upper probabilities.

**Return type**

None

**intervalToCredalWithFiles()**

**Warning:** Deprecated : use intervalToCredal (lrsWrapper with no input / output files needed).

Computes the vertices of each credal set according to their interval definition (uses lrs).

Use this method when using a single BayesNet storing counts of events.

**Return type**

None

**isSeparatelySpecified()**

**Returns**

True if this CredalNet is separately and interval specified, False otherwise.

**Return type**

bool

**lagrangeNormalization()**

Normalize counts of a BayesNet storing counts of each events such that no probability is 0.

Use this method when using a single BayesNet storing counts of events. Lagrange normalization. This call is irreversible and modify counts stored by `__src_bn`.

Does not performs computations of the parameters but keeps normalized counts of events only. Call `idmLearning` to compute the probabilities (with any parameter value).

**Return type**

None

**nodeType(id)**

**Parameters**

**id** (*int*) – the constant reference to the choosen NodeId

**Returns**

the type of the choosen node in the (up-to-date) CredalNet in `__src_bn`.

**Return type**

[\*pyAgrum.CredalNet\*](#) (page 204)

**saveBNsMinMax(min\_path, max\_path)**

If this CredalNet was built over a perturbed BayesNet, one can save the intervals as two BayesNet.

to call after `bnToCredal(GUM_SCALAR beta)` save a BN with lower probabilities and a BN with upper ones

**Parameters**

- **min\_path** (*str*) – the path to save the BayesNet which contains the lower probabilities of each node X.
- **max\_path** (*str*) – the path to save the BayesNet which contains the upper probabilities of each node X.

**Return type**

None

**setCPT(\*args)**

**Warning:** (experimental function) - Parameters to be wrapped

Set the vertices of one credal set of a given node (any instantiation index)

**Parameters**

- **id** (*int*) – the Id of the node
- **entry** (*int*) – the index of the instantiation (from 0 to K - 1) excluding the given node (only the parents are used to compute the index of the credal set)
- **ins** ([\*pyAgrum.Instantiation\*](#) (page 47)) – the Instantiation (only the parents matter to find the credal set index)
- **cpt** (*tbw*) – the vertices of every credal set (for each instantiation of the parents)

**Warning:** DOES not change the BayesNet(s) associated to this credal net !

**Return type**

None

**setCPTs**(*id*, *cpt*)

**Warning:** (experimental function) - Parameters to be wrapped

Set the vertices of the credal sets (all of the conditionals) of a given node

**Parameters**

- **id** (*int*) – the NodeId of the node
- **cpt** (*tbw*) – the vertices of every credal set (for each instantiation of the parents)

**Warning:** DOES not change the BayesNet (s) associated to this credal net !

**src\_bn**()

**Returns**

Returns a constant reference to the original BayesNet (used as a DAG, it's CPTs does not matter).

**Return type**

*pyAgrum.BayesNet* (page 64)

## 1.6.2 CN Inference

**class** *pyAgrum.CNMonteCarloSampling*(*credalNet*)

Class used for inferences in credal networks with Monte Carlo sampling algorithm.

**CNMonteCarloSampling**(*cn*) -> **CNMonteCarloSampling**

**Parameters:**

- **cn** (*pyAgrum.CredalNet*) – a credal network

**Parameters**

**credalNet** (*CredalNet* (page 204)) –

**CN**()

**Return type**

*CredalNet* (page 204)

**currentTime**()

**Returns**

get the current running time in second (float)

**Return type**

float

**dynamicExpMax**(*varName*)

Get the upper dynamic expectation of a given variable prefix.

**Parameters**

**varName** (*str*) – the variable name prefix which upper expectation we want.

**Returns**

a constant reference to the variable upper expectation over all time steps.

**Return type**

float

**dynamicExpMin**(*varName*)

Get the lower dynamic expectation of a given variable prefix.

**Parameters**

**varName** (*str*) – the variable name prefix which lower expectation we want.

**Returns**

a constant reference to the variable lower expectation over all time steps.

**Return type**

float

**epsilon**()**Returns**

the value of epsilon

**Return type**

float

**history**()**Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**insertEvidenceFile**(*path*)

Insert evidence from file.

**Parameters**

**path** (*str*) – the path to the evidence file.

**Return type**

None

**insertModalsFile**(*path*)

Insert variables modalities from file to compute expectations.

**Parameters**

**path** (*str*) – The path to the modalities file.

**Return type**

None

**makeInference**()

Starts the inference.

**Return type**

None

**marginalMax**(\**args*)

Get the upper marginals of a given node id.

**Parameters**

- **id** (*int*) – the node id which upper marginals we want.
- **varName** (*str*) – the variable name which upper marginals we want.

**Returns**

a constant reference to this node upper marginals.

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Credal network

**marginalMin**(\**args*)

Get the lower marginals of a given node id.

**Parameters**

- **id** (*int*) – the node id which lower marginals we want.
- **varName** (*str*) – the variable name which lower marginals we want.

**Returns**

a constant reference to this node lower marginals.

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Credal network

**maxIter()****Returns**

the criterion on number of iterations

**Return type**

int

**maxTime()****Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme()****Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate()****Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**setEpsilon(*eps*)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{eps} < 0$

**Return type**

None

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $\text{max} \leq 1$

**Return type**

None

**setMaxTime**(*timeout*)

**Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If timeout<=0.0

**Return type**

None

**setMinEpsilonRate**(*rate*)

**Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize**(*p*)

**Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[pyAgrum.OutOfBounds](#) (page 290) – If p<1

**Return type**

None

**setRepetitiveInd**(*flag*)

**Parameters**

**flag** (*bool*) – True if repetitive independence is to be used, false otherwise.  
Only usefull with dynamic networks.

**Return type**

None

**setVerbosity**(*v*)

**Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**verbosity**()

**Returns**

True if the verbosity is enabled

**Return type**

bool

**class** `pyAgrum.CNLoopyPropagation`(*cnet*)

Class used for inferences in credal networks with Loopy Propagation algorithm.

**CNLoopyPropagation**(*cn*) -> **CNLoopyPropagation**

**Parameters:**

- **cn** (*pyAgrum.CredalNet*) – a Credal network

**Parameters**

**cnet** (*CredalNet* (page 204)) –

**CN**()

**Return type**

*CredalNet* (page 204)

**InferenceType\_nodeToNeighbours** = 0

**InferenceType\_ordered** = 1



**InferenceType\_randomOrder** = 2

**currentTime()**

**Returns**

get the current running time in second (float)

**Return type**

float

**dynamicExpMax**(*varName*)

Get the upper dynamic expectation of a given variable prefix.

**Parameters**

**varName** (*str*) – the variable name prefix which upper expectation we want.

**Returns**

a constant reference to the variable upper expectation over all time steps.

**Return type**

float

**dynamicExpMin**(*varName*)

Get the lower dynamic expectation of a given variable prefix.

**Parameters**

**varName** (*str*) – the variable name prefix which lower expectation we want.

**Returns**

a constant reference to the variable lower expectation over all time steps.

**Return type**

float

**epsilon()**

**Returns**

the value of epsilon

**Return type**

float

**eraseAllEvidence()**

Erase all inference related data to perform another one.

You need to insert evidence again if needed but modalities are kept. You can insert new ones by using the appropriate method which will delete the old ones.

**Return type**

None

**history()**

**Returns**

the scheme history

**Return type**

tuple

**Raises**

[\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the scheme did not performed or if verbosity is set to false

**inferenceType**(*\*args*)

**Returns**

the inference type

**Return type**

int

**insertEvidenceFile**(*path*)

Insert evidence from file.

**Parameters**

**path** (*str*) – the path to the evidence file.

**Return type**

None

**insertModalsFile**(*path*)

Insert variables modalities from file to compute expectations.

**Parameters**

**path** (*str*) – The path to the modalities file.

**Return type**

None

**makeInference**()

Starts the inference.

**Return type**

None

**marginalMax**(\**args*)

Get the upper marginals of a given node id.

**Parameters**

- **id** (*int*) – the node id which upper marginals we want.
- **varName** (*str*) – the variable name which upper marginals we want.

**Returns**

a constant reference to this node upper marginals.

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Credal network

**marginalMin**(\**args*)

Get the lower marginals of a given node id.

**Parameters**

- **id** (*int*) – the node id which lower marginals we want.
- **varName** (*str*) – the variable name which lower marginals we want.

**Returns**

a constant reference to this node lower marginals.

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Credal network

**maxIter**()**Returns**

the criterion on number of iterations

**Return type**

int

**maxTime**()**Returns**

the timeout(in seconds)

**Return type**

float

**messageApproximationScheme**()**Returns**

the approximation scheme message

**Return type**

str

**minEpsilonRate**()**Returns**

the value of the minimal epsilon rate

**Return type**

float

**nbrIterations()****Returns**

the number of iterations

**Return type**

int

**periodSize()****Returns**

the number of samples between 2 stopping

**Return type**

int

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**saveInference(*path*)**

Saves marginals.

**Parameters**

**path** (*str*) – The path to the file to save marginals.

**Return type**

None

**setEpsilon(*eps*)****Parameters**

**eps** (*float*) – the epsilon we want to use

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $eps < 0$

**Return type**

None

**setMaxIter(*max*)****Parameters**

**max** (*int*) – the maximum number of iteration

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $max \leq 1$

**Return type**

None

**setMaxTime(*timeout*)****Parameters**

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (*float*) –

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $timeout \leq 0.0$

**Return type**

None

**setMinEpsilonRate(*rate*)****Parameters**

**rate** (*float*) – the minimal epsilon rate

**Return type**

None

**setPeriodSize(*p*)****Parameters**

**p** (*int*) – number of samples between 2 stopping

**Raises**

[\*pyAgrum.OutOfBounds\*](#) (page 290) – If  $p < 1$

**Return type**

None

**setRepetitiveInd**(*flag*)

**Parameters**

**flag** (*bool*) – True if repetitive independence is to be used, false otherwise.  
Only usefull with dynamic networks.

**Return type**

None

**setVerbosity**(*v*)

**Parameters**

**v** (*bool*) – verbosity

**Return type**

None

**property thisown**

The membership flag

**verbosity**()

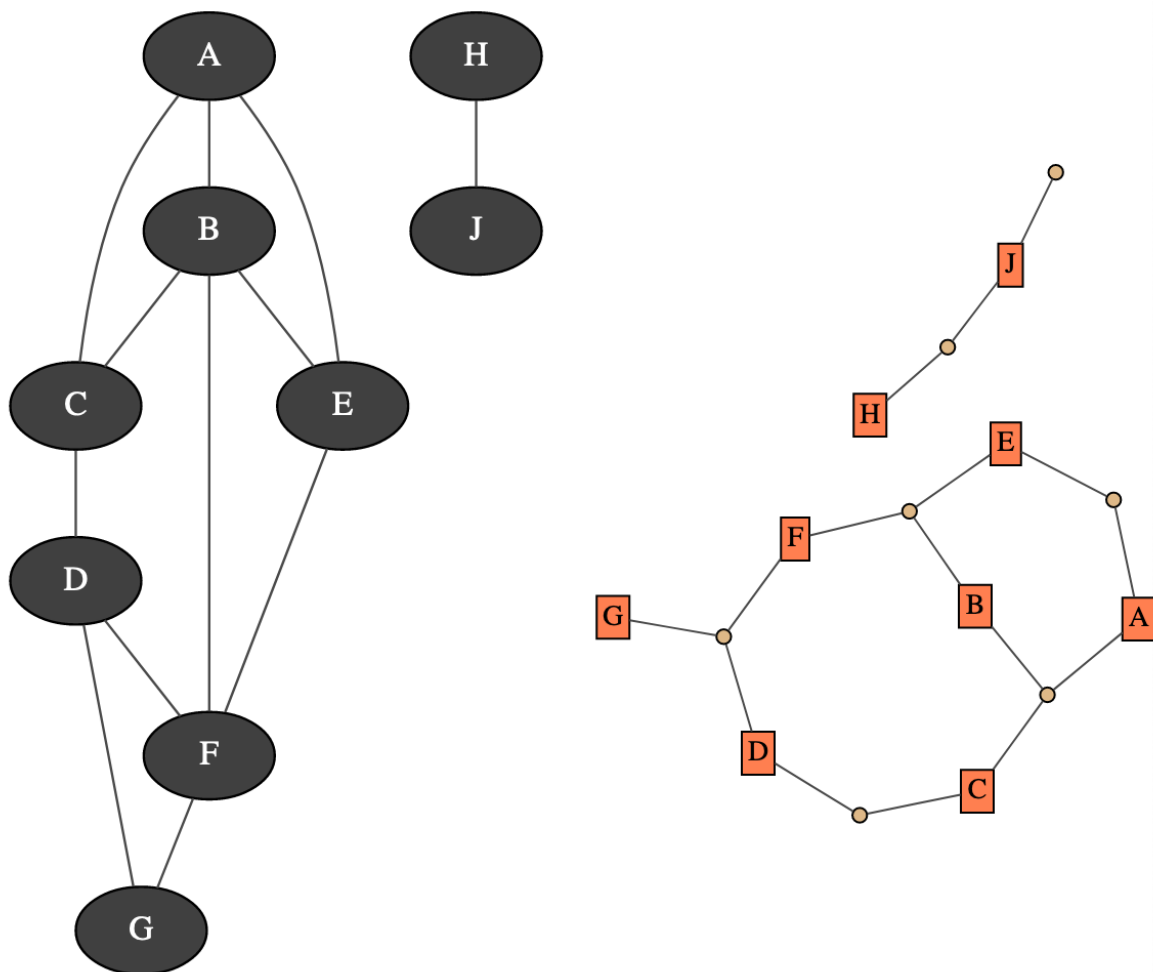
**Returns**

True if the verbosity is enabled

**Return type**

bool

## 1.7 Markov Network



A Markov network is a undirected probabilistic graphical model. It represents a joint distribution over a set of random variables. In pyAgrum, the variables are (for now) only discrete.

A Markov network uses a undirected graph to represent conditional independence in the joint distribution. These conditional independence allow to factorize the joint distribution, thereby allowing to compactly represent very large ones.

$$P(X_1, \dots, X_n) \propto \prod_{i=1}^{n_c} \phi_i(C_i)$$

Where the  $\phi_i$  are potentials over the  $n_c$  cliques of the undirected graph.

Moreover, inference algorithms can also use this graph to speed up the computations.

### Tutorial

- [Tutorial on Markov Network](#) (page 369)

### Reference

## 1.7.1 Undirected Graphical Model

**class** pyAgrum.**MarkovNet**(\*args)

MarkovNet represents a Markov Network.

**MarkovNet**(name='') -> **MarkovNet**

#### Parameters:

- **name** (*str*) – the name of the Bayes Net

**MarkovNet**(source) -> **MarkovNet**

#### Parameters:

- **source** (*pyAgrum.MarkovNet*) – the Markov network to copy

**add**(\*args)

Add a variable to the pyAgrum.MarkovNet.

#### Parameters

- **variable** (*pyAgrum.DiscreteVariable* (page 25)) – the variable added
- **name** (*str*) – the variable name
- **nbrmod** (*int*) – the number of modalities for the new variable
- **id** (*int*) – the variable forced id in the pyAgrum.MarkovNet

#### Returns

the id of the new node

#### Return type

int

#### Raises

- **pyAgrum.DuplicateLabel** (page 287) – If variable.name() is already used in this pyAgrum.MarkovNet.
- **pyAgrum.NotAllowed** – If nbrmod is less than 2
- **pyAgrum.DuplicateElement** (page 287) – If id is already used.

**addFactor**(\*args)

Add a factor from a list or a set of id or str. If the argument is a set, the order is the order of the IDs of the variables

#### Parameters

**seq** (*sequence (list or set) of int or string*) – The sequence (ordered or not) of node id or names

#### Return type

*Potential* (page 53)

**addStructureListener**(*whenNodeAdded=None, whenNodeDeleted=None, whenEdgeAdded=None, whenEdgeDeleted=None*)

Add the listeners in parameters to the list of existing ones.

**Parameters**

- **whenNodeAdded** (*lambda expression*) – a function for when a node is added
- **whenNodeDeleted** (*lambda expression*) – a function for when a node is removed
- **whenEdgeAdded** (*lambda expression*) – a function for when an edge is added
- **whenEdgeDeleted** (*lambda expression*) – a function for when an edge is removed

**addVariables**(*listFastVariables, default\_nbr\_mod=2*)

Add a list of variable in the form of ‘fast’ syntax.

**Parameters**

- **listFastVariables** (*List[str]*) – the list of variables in ‘fast’ syntax.
- **default\_nbr\_mod** (*int*) – the number of modalities for the variable if not specified following *fast syntax* (page 279). Note that `default_nbr_mod=1` is mandatory to create variables with only one modality (for utility for instance).

**Returns**

the list of created ids.

**Return type**

List[int]

**beginTopologyTransformation()**

**Return type**

None

**changeVariableLabel**(*\*args*)

change the label of the variable associated to `nodeId` to the new value.

**Parameters**

- **var** (*Union[int, str]*) – a variable’s id (int) or name
- **old\_label** (*str*) – the old label
- **new\_label** (*str*) – the new label

**Raises**

[\*pyAgrum.NotFound\*](#) (page 289) – if id/name is not a variable or if `old_label` does not exist.

**Return type**

None

**changeVariableName**(*\*args*)

Changes a variable’s name in the `pyAgrum.MarkovNet`.

This will change the “`pyAgrum.DiscreteVariable`” names in the `pyAgrum.MarkovNet`.

**Parameters**

- **car** (*Union[int, str]*) – a variable’s id (int) or name
- **new\_name** (*str*) – the new name of the variable

**Raises**

- [\*pyAgrum.DuplicateLabel\*](#) (page 287) – If `new_name` is already used in this `MarkovNet`.
- [\*pyAgrum.NotFound\*](#) (page 289) – If no variable matches id.

**Return type**

None

**clear()**

Clear the whole `MarkovNet`

**Return type**

None

**completeInstantiation()****Return type**[Instantiation](#) (page 47)**connectedComponents()**

connected components from a graph/BN

Compute the connected components of a pyAgrum's graph or Bayesian Network (more generally an object that has *nodes*, *children/parents* or *neighbours* methods)

The firstly visited node for each component is called a 'root' and is used as a key for the component. This root has been arbitrarily chosen during the algorithm.

**Returns**

dict of connected components (as set of nodeIds (int)) with a nodeId (root) of each component as key.

**Return type**

dict(int,Set[int])

**dim()****Return type**

int

**edges()****Return type**

object

**empty()****Return type**

bool

**endTopologyTransformation()**

Terminates a sequence of insertions/deletions of arcs by adjusting all CPTs dimensions. End Multiple Change for all CPTs.

**Return type**

[pyAgrum.MarkovNet](#) (page 217)

**erase(\*args)**

Remove a variable from the pyAgrum.MarkovNet.

Removes the corresponding variable from the pyAgrum.MarkovNet and from all of it's children pyAgrum.Potential.

If no variable matches the given id, then nothing is done.

**Parameters**

**var** ([Union\[int,str,pyAgrum.DiscreteVariable](#) (page 25)] – a variable's id (int) or name of variable or a reference of this variable to remove.

**Return type**

None

**eraseFactor(\*args)****Return type**

None

**exists(node)****Parameters**

**node** (int) –

**Return type**

bool

**existsEdge(\*args)****Return type**

bool

**factor(\*args)**

Returns the factor of a set of variables (if existing).

**Parameters**

**vars** (*Union[Set[int], Set[str]]*) – A set of ids or names of variable the pyAgrum.MarkovNet.

**Returns**

The factor of the set of nodes.

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

*pyAgrum.NotFound* (page 289) – If no variable's id matches varId.

**factors()****Return type**

List[Set[int]]

**static fastPrototype(dotlike, domainSize=2)**

Create a Markov network with a modified dot-like syntax which specifies:

- the structure a-b-c;b-d-e;. The substring a-b-c indicates a factor with the scope (a,b,c).
- the type of the variables with different syntax (cf documentation).

**Examples**

```
>>> import pyAgrum as gum
>>> bn=pyAgrum.MarkovNet.fastPrototype('A--B[1,3]-C{yes|No}--D[2,4]--
↳ E[1,2.5,3.9]',6)
```

**Parameters**

- **dotlike** (*str*) – the string containing the specification
- **domainSize** (*int*) – the default domain size for variables

**Returns**

the resulting Markov network

**Return type**

*pyAgrum.MarkovNet* (page 217)

**static fromBN(bn)****Parameters**

**bn** (*BayesNet* (page 64)) –

**Return type**

*MarkovNet* (page 217)

**generateFactor(vars)**

Randomly generate factor parameters for a given factor in a given structure.

**Parameters**

- **node** (*Union[int, str]*) – a variable's id (int) or name
- **vars** (*List[int]*) –

**Return type**

None

**generateFactors()**

Randomly generates factors parameters for a given structure.

**Return type**

None

**graph()****Return type**

*UndiGraph* (page 12)



**hasSameStructure**(*other*)

**idFromName**(*name*)

**Parameters**

**name** (*str*) –

**Return type**

int

**ids**(*names*)

**isIndependent**(\**args*)

**Return type**

bool

**loadUAI**(\**args*)

Load an UAI file.

**Parameters**

- **name** (*str*) – the name's file
- **l** (*list*) – list of functions to execute

**Raises**

- [\*pyAgrum.IOError\*](#) (page 288) – If file not found
- [\*pyAgrum.FatalError\*](#) (page 287) – If file is not valid

**Return type**

str

**log10DomainSize**()

**Return type**

float

**maxNonOneParam**()

**Return type**

float

**maxParam**()

**Return type**

float

**maxVarDomainSize**()

**Return type**

int

**minNonZeroParam**()

**Return type**

float

**minParam**()

**Return type**

float

**minimalCondSet**(\**args*)

**Return type**

object

**names**()

**Returns**

The names of the graph variables

**Return type**

List[str]

**neighbours**(*norid*)

**Parameters**

**norid** (object) –

**Return type**  
object

**nodeId**(*var*)

**Parameters**  
**var** (*DiscreteVariable* (page 25)) –

**Return type**  
int

**nodes**()

**Return type**  
object

**nodeset**(*names*)

**Parameters**  
**names** (Vector\_string) –

**Return type**  
List[int]

**saveUAI**(*name*)

Save the MarkovNet in an UAI file.

**Parameters**  
**name** (*str*) – the file's name

**Return type**  
None

**size**()

**Return type**  
int

**sizeEdges**()

**Return type**  
int

**smallestFactorFromNode**(*node*)

**Parameters**  
**node** (int) –

**Return type**  
List[int]

**property thisown**

The membership flag

**toDot**()

**Return type**  
str

**toDotAsFactorGraph**()

**Return type**  
str

**variable**(\**args*)

**Return type**  
*DiscreteVariable* (page 25)

**variableFromName**(*name*)

**Parameters**  
**name** (*str*) –

**Return type**  
*DiscreteVariable* (page 25)

**variableNodeMap**()

## 1.7.2 Inference in Markov Networks

Inference is the process that consists in computing new probabilistic information from a Markov network and some evidence. aGrUM/pyAgrum mainly focus and the computation of (joint) posterior for some variables of the Markov networks given soft or hard evidence that are the form of likelihoods on some variables. Inference is a hard task (NP-complete). For now, aGrUM/pyAgrum implements only one exact inference for Markov Network.

### Shafer Shenoy Inference in Markov Network

**class** pyAgrum.**ShaferShenoyMNIInference**(*MN*, *use\_binary\_join\_tree=True*)

Class used for Shafer-Shenoy inferences for Markov network.

**ShaferShenoyInference**(*bn*) -> **ShaferShenoyInference**

**Parameters:**

- **mn** (*pyAgrum.MarkovNet*) – a Markov network

**Parameters**

- **MN** (*IMarkovNet*) –
- **use\_binary\_join\_tree** (*bool*) –

**H**(\**args*)

**Parameters**

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

the computed Shanon's entropy of a node given the observation

**Return type**

float

**I**(*X*, *Y*)

**Parameters**

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

**Returns**

- -----
- **float** – the Mutual Information of X and Y given the observation

**Return type**

float

**MN**()

**VI**(*X*, *Y*)

**Parameters**

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

**Returns**

- -----
- **float** – variation of information between X and Y

**Return type**

float

**addAllTargets**()

Add all the nodes as targets.

**Return type**

None

**addEvidence(\*args)**

Adds a new evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** – (*int*) a node value
- **val** – (*str*) the label of the node value
- **vals** (*list*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node already has an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**addJointTarget(targets)**

Add a list of nodes as a new joint target. As a collateral effect, every node is added as a marginal target.

**Parameters**

- **list** – a list of names of nodes
- **targets** (object) –

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If some node(s) do not belong to the Bayesian network

**Return type**

None

**addTarget(\*args)**

Add a marginal target to the list of targets.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Raises**

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If target is not a NodeId in the Bayes net

**Return type**

None

**chgEvidence(\*args)**

Change the value of an already existing evidence on a node (might be soft or hard).

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name
- **val** (*intstr*) – a node value or the label of the node value
- **vals** (*List[float]*) – a list of values

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the node does not already have an evidence
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If val is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of vals is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If vals is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If the node does not belong to the Bayesian network

**Return type**

None

**eraseAllEvidence()**

Removes all the evidence entered into the network.

**Return type**

None

**eraseAllJointTargets()**

Clear all previously defined joint targets.

**Return type**

None

**eraseAllMarginalTargets()**

Clear all the previously defined marginal targets.

**Return type**

None

**eraseAllTargets()**

Clear all previously defined targets (marginal and joint targets).

As a result, no posterior can be computed (since we can only compute the posteriors of the marginal or joint targets that have been added by the user).

**Return type**

None

**eraseEvidence(\*args)**

Remove the evidence, if any, corresponding to the node Id or name.

**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**Return type**

None

**eraseJointTarget(targets)**

Remove, if existing, the joint target.

**Parameters**

- **list** – a list of names or Ids of nodes
- **targets** (object) –

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**eraseTarget(\*args)**

Remove, if existing, the marginal target.

**Parameters**

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

**Raises**

- **pyAgrum.IndexError** – If one of the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**Return type**

None

**evidenceImpact**(*target, evs*)

Create a pyAgrum.Potential for P(target|evs) (for all instantiation of target and evs)

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns**

a Potential for P(targets|evs)

**Return type**

*pyAgrum.Potential* (page 53)

**evidenceJointImpact**(*\*args*)

Create a pyAgrum.Potential for P(joint targets|evs) (for all instantiation of targets and evs)

**Parameters**

- **targets** (*List[intstr]*) – a list of node Ids or node names
- **evs** (*Set[intstr]*) – a set of nodes ids or names.

**Returns**

a Potential for P(target|evs)

**Return type**

*pyAgrum.Potential* (page 53)

**Raises**

**pyAgrum.Exception** – If some evidence entered into the Bayes net are incompatible (their joint proba = 0)

**evidenceProbability**()**Returns**

the probability of evidence

**Return type**

float

**getNumberOfThreads**()

returns the number of threads used by LazyPropagation during inferences.

**Returns**

the number of threads used by LazyPropagation during inferences

**Return type**

int

**hardEvidenceNodes**()**Returns**

the set of nodes with hard evidence

**Return type**

set

**hasEvidence**(*\*args*)**Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if some node(s) (or the one in parameters) have received evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasHardEvidence**(*nodeName*)**Parameters**

- **id** (*int*) – a node Id

- **nodeName** (*str*) – a node name

**Returns**

True if node has received a hard evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**hasSoftEvidence(\*args)****Parameters**

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if node has received a soft evidence

**Return type**

bool

**Raises**

**pyAgrum.IndexError** – If the node does not belong to the Bayesian network

**isGumNumberOfThreadsOverridden()**

Indicates whether LazyPropagation currently overrides aGrUM's default number of threads (see method `setNumberOfThreads`).

**Returns**

A Boolean indicating whether LazyPropagation currently overrides aGrUM's default number of threads

**Return type**

bool

**isJointTarget(targets)****Parameters**

- **list** – a list of nodes ids or names.
- **targets** (object) –

**Returns**

True if target is a joint target.

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**isTarget(\*args)****Parameters**

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

**Returns**

True if variable is a (marginal) target

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the node does not belong to the Bayesian network
- **pyAgrum.UndefinedElement** (page 291) – If node Id is not in the Bayesian network

**joinTree()****Returns**

the current join tree used

**Return type**

*pyAgrum.CliqueGraph* (page 15)

**jointMutualInformation**(*targets*)

**Parameters**

**targets** (object) –

**Return type**

float

**jointPosterior**(*targets*)

Compute the joint posterior of a set of nodes.

**Parameters**

**list** – the list of nodes whose posterior joint probability is wanted

**Warning:** The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used by the Potential.

**Returns**

a const ref to the posterior joint probability of the set of nodes.

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**Parameters**

**targets** (object) –

**jointTargets**()

**Returns**

the list of target sets

**Return type**

list

**junctionTree**()

**Returns**

the current junction tree

**Return type**

[\*pyAgrum.CliqueGraph\*](#) (page 15)

**makeInference**()

Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

**Return type**

None

**nbrEvidence**()

**Returns**

the number of evidence entered into the Bayesian network

**Return type**

int

**nbrHardEvidence**()

**Returns**

the number of hard evidence entered into the Bayesian network

**Return type**

int

**nbrJointTargets**()



**Returns**

the number of joint targets

**Return type**

int

**nbrSoftEvidence()****Returns**

the number of soft evidence entered into the Bayesian network

**Return type**

int

**nbrTargets()****Returns**

the number of marginal targets

**Return type**

int

**posterior(\*args)**

Computes and returns the posterior of a node.

**Parameters**

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

**Returns**

a const ref to the posterior probability of the node

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If an element of nodes is not in targets

**setEvidence(evidces)**

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain size of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

**setMaxMemory(gigabytes)**

sets an upper bound on the memory consumption admissible

**Parameters**

**gigabytes** (*float*) – this upper bound in gigabytes.

**Return type**

None

**setNumberOfThreads(nb)**

If the argument nb is different from 0, this number of threads will be used during inferences, hence overriding aGrUM's default number of threads. If, on the contrary, nb is equal to 0, the parallelized inference engine will comply with aGrUM's default number of threads.

**Parameters**

**nb** (*int*) – the number of threads to be used by ShaferShenoyMNIInference

**Return type**

None

**setTargets(*targets*)**

Remove all the targets and add the ones in parameter.

**Parameters**

**targets** (*set*) – a set of targets

**Raises**

[\*pyAgrum.UndefinedElement\*](#) (page 291) – If one target is not in the Bayes net

**softEvidenceNodes()****Returns**

the set of nodes with soft evidence

**Return type**

set

**targets()****Returns**

the list of marginal targets

**Return type**

list

**property thisown**

The membership flag

**updateEvidence(*evidces*)**

Apply `chgEvidence(key,value)` for every pairs in `evidces` (or `addEvidence`).

**Parameters**

**evidces** (*dict*) – a dict of evidences

**Raises**

- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If one value is not a value for the node
- [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the size of a value is different from the domain side of the node
- [\*pyAgrum.FatalError\*](#) (page 287) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

## 1.8 Probabilistic Relational Models

For now, pyAgrum only allows to explore Probabilistic Relational Models written with o3prm syntax (see [O3PRM website](https://o3prm.gitlab.io/) (<https://o3prm.gitlab.io/>)).

**class pyAgrum.PRMexplorer**

PRMexplorer helps navigate through probabilistic relational models.

**PRMexplorer() -> PRMexplorer**

default constructor

**property aggType**

min/max/count/exists/forall/or/and/amplitude/median

**classAggregates(*class\_name*)****Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of aggregates in the class

**Return type**

list

**Raises**

[\*pyAgrum.IndexError\*](#) – If the class is not in the PRM

**classAttributes**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of attributes

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the class is not in the PRM

**classDag**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

a description of the DAG

**Return type**

tuple

**Raises**

**pyAgrum.IndexError** – If the class is not in the PRM

**classImplements**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of interfaces implemented by the class

**Return type**

list

**classParameters**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of parameters

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the class is not in the PRM

**classReferences**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of references

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the class is not in the PRM

**classSlotChains**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of class slot chains

**Return type**

list

**Raises**

**pyAgrum.IndexError** – if the class is not in the PRM

**classes**()

**Returns**

the list of classes

**Return type**

list

**cpf**(*class\_name*, *attribute*)

**Parameters**

- **class\_name** (*str*) – a class name
- **attribute** (*str*) – an attribute

**Returns**

the potential of the attribute

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**Raises**

- [\*pyAgrum.OperationNotAllowed\*](#) (page 290) – If the class element doesn't have any [\*pyAgrum.Potential\*](#) (like a `gum::PRMReferenceSlot`).
- [\*pyAgrum.IndexError\*](#) – If the class is not in the PRM
- [\*pyAgrum.IndexError\*](#) – If the attribute in parameters does not exist

**getDirectSubClass**(*class\_name*)

**Parameters**

**class\_name** (*str*) – a class name

**Returns**

the list of direct subclasses

**Return type**

list

**Raises**

[\*pyAgrum.IndexError\*](#) – If the class is not in the PRM

**getDirectSubInterfaces**(*interface\_name*)

**Parameters**

**interface\_name** (*str*) – an interface name

**Returns**

the list of direct subinterfaces

**Return type**

list

**Raises**

[\*pyAgrum.IndexError\*](#) – If the interface is not in the PRM

**getDirectSubTypes**(*type\_name*)

**Parameters**

**type\_name** (*str*) – a type name

**Returns**

the list of direct subtypes

**Return type**

list

**Raises**

[\*pyAgrum.IndexError\*](#) – If the type is not in the PRM

**getImplementations**(*interface\_name*)

**Parameters**

**interface\_name** (*str*) – an interface name

**Returns**

the list of classes implementing the interface

**Return type**

str

**Raises**

[\*pyAgrum.IndexError\*](#) – If the interface is not in the PRM

**getLabelMap**(*type\_name*)

**Parameters****type\_name** (*str*) – a type name**Returns**

a dict containing pairs of label and their values

**Return type**

dict

**Raises****pyAgrum.IndexError** – If the type is not in the PRM**getLabels**(*type\_name*)**Parameters****type\_name** (*str*) – a type name**Returns**

the list of type labels

**Return type**

list

**Raises****pyAgrum.IndexError** – If the type is not in the PRM**getSuperClass**(*class\_name*)**Parameters****class\_name** (*str*) – a class name**Returns**

the class extended by class\_name

**Return type**

str

**Raises****pyAgrum.IndexError** – If the class is not in the PRM**getSuperInterface**(*interface\_name*)**Parameters****interface\_name** (*str*) – an interface name**Returns**

the interace extended by interface\_name

**Return type**

str

**Raises****pyAgrum.IndexError** – If the interface is not in the PRM**getSuperType**(*type\_name*)**Parameters****type\_name** (*str*) – a type name**Returns**

the type extended by type\_name

**Return type**

str

**Raises****pyAgrum.IndexError** – If the type is not in the PRM**getalltheSystems**()**Returns**

the list of all the systems and their components

**Return type**

list

**interAttributes**(*interface\_name*, *allAttributes=False*)**Parameters**

- **interface\_name** (*str*) – an interface
- **allAttributes** (*bool*) – True if supertypes of a custom type should be indicated

**Returns**

the list of (<type>,<attribute\_name>) for the given interface

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the type is not in the PRM

**interReferences**(*interface\_name*)**Parameters**

**interface\_name** (*str*) – an interface

**Returns**

the list of (<reference\_type>,<reference\_name>,<True if the reference is an array>) for the given interface

**Return type**

list

**Raises**

**pyAgrum.IndexError** – If the type is not in the PRM

**interfaces**()**Returns**

the list of interfaces in the PRM

**Return type**

list

**isAttribute**(*class\_name*, *att\_name*)**Parameters**

- **class\_name** (*str*) – a class name
- **att\_name** (*str*) – the name of the attribute to be tested

**Returns**

True if att\_name is an attribute of class\_name

**Return type**

bool

**Raises**

- **pyAgrum.IndexError** – If the class is not in the PRM
- **pyAgrum.IndexError** – If att\_name is not an element of class\_name

**isClass**(*name*)**Parameters**

**name** (*str*) – an element name

**Returns**

True if the parameter correspond to a class in the PRM

**Return type**

bool

**isInterface**(*name*)**Parameters**

**name** (*str*) – an element name

**Returns**

True if the parameter correspond to an interface in the PRM

**Return type**

bool

**isType**(*name*)**Parameters**

**name** (*str*) – an element name

**Returns**

True if the parameter correspond to a type in the PRM

**Return type**

bool

**load(\*args)**

Load a PRM into the explorer.

**Parameters**

- **filename** (*str*) – the name of the o3prm file
- **classpath** (*str*) – the classpath of the PRM

**Raises**

[\*pyAgrum.FatalError\*](#) (page 287) – If file not found

**Return type**

None

**types()**

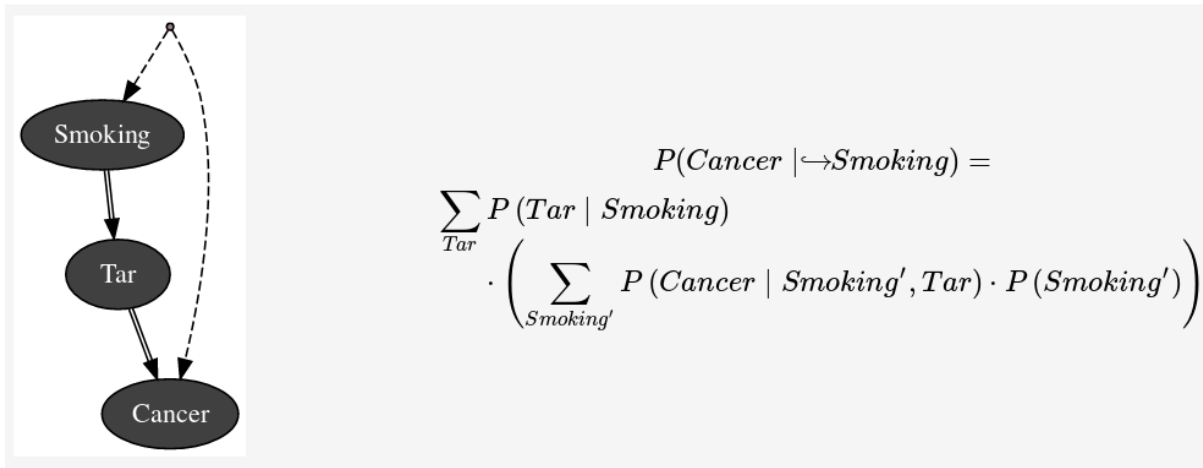
**Returns**

the list of the custom types in the PRM

**Return type**

list

## 1.9 pyAgrum.causal documentation



Causality in pyAgrum mainly consists in the ability to build a causal model, i.e. a (observational) Bayesian network and a set of latent variables and their relation with observation variables and in the ability to compute using do-calculus the causal impact in such a model.

Causality is a set of pure python3 scripts based on pyAgrum's tools.

**Note:** As it can be seen in the figure above, *pyAgrum.causal* module uses a LaTeX special arrow ( $\hookrightarrow$ ) to compactly represent an intervention. If you prefer the classical “do” notation, you can change this behavior by using:

```
gum.config["causal","latex_do_prefix"]="do("
gum.config["causal","latex_do_suffix"]=")"
```

### Tutorials

- [Notebooks on causality in pyAgrum.](#)
- Some [implemented examples](#) (<https://webia.lip6.fr/~phw/aGrUM/BookOfWhy/>) from the [book of Why](#) (<http://bayes.cs.ucla.edu/WHY/>) from Judea Pearl and Dana Mackenzie.

### Reference

### 1.9.1 Causal Model

**class** `pyAgrum.causal.CausalModel`(*bn, latentVarsDescriptor=None, keepArcs=False*)

From an observational BNs and the description of latent variables, this class represent a complet causal model obtained by adding the latent variables specified in `latentVarsDescriptor` to the Bayesian network `bn`.

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) – an observational Bayesian network
- **latentVarsDescriptor** (`List[(str, List[int])]`) – list of couples (<latent variable name>, <list of affected variables' ids>).
- **keepArcs** (`bool`) – By default, the arcs between variables affected by a common latent variable will be removed but this can be avoided by setting `keepArcs` to `True`

**addCausalArc**(*x, y*)

Add an arc `x->y`

#### Parameters

- **x** (`int / str`) – the `nodeId` or the name of the first node
- **y** (`int / str`) – the `nodeId` or the name of the second node

#### Return type

`None`

**addLatentVariable**(*name, lchild, keepArcs=False*)

Add a new latent variable with a name, a tuple of children and replacing (or not) correlations between children.

#### Parameters

- **name** (`str`) – the name of the latent variable
- **lchild** (`Tuple[str, str]`) – the tuple of (2) children
- **keepArcs** (`bool`) – do we keep (or not) the arc between the children ?

#### Return type

`None`

**arcs**()

#### Return type

`Set[Tuple[NewType()(NodeId, int), NewType()(NodeId, int)]]`

#### Returns

the set of arcs

**backDoor**(*cause, effect, withNames=True*)

Check if a backdoor exists between *cause* and *effect*

#### Parameters

- **cause** (`int / str`) – the `nodeId` or the name of the cause
- **effect** (`int / str`) – the `nodeId` or the name of the effect
- **withNames** (`bool`) – wether we use ids (int) or names (str)

#### Returns

`None` if no found backdoor. Otherwise return the found backdoors as set of ids or set of names.

#### Return type

`None|Set[str]|Set[int]`

**causalBN**()

#### Return type

`BayesNet` (page 64)

#### Returns

the causal Bayesian network

#### Warning

do not infer any computations in this model. It is strictly a structural model



**children(*x*)**

From a NodeId, returns its children (as a set of NodeId)

**Parameters**

**x** (*int*) – the node

**Returns**

the set of children

**Return type**

Set[int]

**eraseCausalArc(*x*, *y*)**

Erase the arc *x*->*y*

**Parameters**

• **x** (*int* / *str*) – the nodeId or the name of the first node

• **y** (*int* / *str*) – the nodeId or the name of the second node

**Return type**

None

**existsArc(*x*, *y*)**

Does the arc *x*->*y* exist ?

**Parameters**

• **x** (*int* / *str*) – the nodeId or the name of the first node

• **y** (*int* / *str*) – the nodeId or the name of the second node

**Returns**

True if the arc exists.

**Return type**

bool

**frontDoor(*cause*, *effect*, *withNames*=True)**

Check if a frontdoor exists between cause and effet

**Parameters**

• **cause** (*int* / *str*) – the nodeId or the name of the cause

• **effect** (*int* / *str*) – the nodeId or the name of the effect

• **withNames** (*bool*) – wether we use ids (int) or names (str)

**Returns**

None if no found frontdoot. Otherwise return the found frontdoors as set of ids or set of names.

**Return type**

None|Set[str]|Set[int]

**idFromName(*name*)****Parameters**

**name** (*str*) – the name of the variable

**Returns**

the id of the variable

**Return type**

int

**latentVariablesIds()****Returns**

the set of ids of latent variables in the causal model

**Return type**

NodeSet

**names()****Returns**

the map NodeId,Name

**Return type**

Dict[int,str]

**nodes()**  
**Return type**  
Set[NewType()(NodeId, int)]  
**Returns**  
the set of nodes

**observationalBN()**  
**Return type**  
[BayesNet](#) (page 64)  
**Returns**  
the observational Bayesian network

**parents(x)**  
From a NodeId, returns its parent (as a set of NodeId)  
**Parameters**  
**x** (*int*) – the node  
**Returns**  
the set of parents  
**Return type**  
Set[int]

**toDot()**  
Create a dot representation of the causal model  
**Return type**  
str  
**Returns**  
the dot representation in a string

## 1.9.2 Causal Formula

*CausalFormula* is the class that represents a causal query in a causal model. Mainly it consists in

- a reference to the CausalModel
- Three sets of variables name that represent the 3 sets of variable in the query  $P(\text{set1} \mid \text{doing}(\text{set2}), \text{knowing}(\text{set3}))$ .
- the AST for compute the query.

**class** pyAgrum.causal.CausalFormula(*cm, root, on, doing, knowing=None*)

Represents a causal query in a causal model. The query is encoded as an CausalFormula that can be evaluated in the causal model :  $\$P(\text{on} \mid \text{knowing}, \text{overhook}(\text{doing}))\$$

**Parameters**

- **cm** ([CausalModel](#) (page 236)) – the causal model
- **root** ([ASTtree](#) (page 241)) – the syntax tree
- **on** (*str/Set[str]*) – the variable or the set of variables of interest
- **doing** (*str/Set[str]*) – the intervention variable(s)
- **knowing** (*None/str/Set[str]*) – the observation variable(s)

**property** **cm**: [CausalModel](#) (page 236)

**Returns**

the causal model

**Return type**

[CausalModel](#) (page 236)

**copy()**

Copy the AST. Note that the causal model is just referenced. The tree is copied.

**Returns**

the copu

**Return type**

*CausalFormula* (page 238)

**eval()**

Compute the Potential from the CausalFormula over vars using cond as value for others variables

**Returns**

The resulting distribution

**Return type**

*pyAgrum.Potential* (page 53)

**latexQuery(values=None)**

Returns a string representing the query compiled by this Formula. If values, the query is annotated with the values in the dictionary.

**Parameters**

**values** (*None* / *Dict[str, str]*) – the values to add in the query representation

**Returns**

the LaTeX representation of the causal query for this CausalFormula

**Return type**

str

**property root: ASTtree (page 241)****Returns**

the causalFormula as an ASTtree

**Return type**

*ASTtree* (page 241)

**toLatex()****Returns**

a LaTeX representation of the CausalFormula

**Return type**

str

### 1.9.3 Causal Inference

Obtaining and evaluating a CausalFormula is done using one these functions :

`pyAgrum.causal.causalImpact(cm, on, doing, knowing=None, values=None)`

Determines the causal impact of interventions.

Determines the causal impact of the interventions specified in **doing** on the single or list of variables **on** knowing the states of the variables in **knowing** (optional). These last parameters is dictionary <variable name>:<value>. The causal impact is determined in the causal DAG **cm**. This function returns a triplet with a latex format formula used to compute the causal impact, a potential representing the probability distribution of **on** given the interventions and observations as parameters, and an explanation of the method allowing the identification. If there is no impact, the joint probability of **on** is simply returned. If the impact is not identifiable the formula and the adjustment will be *None* but an explanation is still given.

**Parameters**

- **cm** (*CausalModel* (page 236)) – the causal model
- **on** (*str* / *NameSet*) – variable name or variable names set of interest
- **doing** (*str* / *NameSet*) – the interventions

- **knowing** (*str/NameSet*) – the observations
- **values** (*Dict[str,int] default=None*) – the values of interventions and observations

**Returns**

the CausalFormula, the computation, the explanation

**Return type**

Tuple[CausalFormula (page 238),pyAgrum.Potential (page 53),str]

pyAgrum.causal.doCalculusWithObservation(*cm, on, doing, knowing=None*)

Compute the CausalFormula for an impact analysis given the causal model, the observed variables and the variable on which there will be intervention.

**Parameters**

- **cm** (CausalModel (page 236)) – the causal model
- **on** (Set[str]) – the variables of interest
- **doing** (Set[str]) – the interventions
- **knowing** (Set[str] default=None) – the observations

**Returns**

if possible, returns the formula to compute this intervention

**Return type**

CausalFormula (page 238)

**Raises**

HedgeException (page 252), UnidentifiableException (page 252) – if this calculus is not possible

pyAgrum.causal.identifyingIntervention(*cm, Y, X, P=None*)

Following Shpitser, Ilya and Judea Pearl. ‘Identification of Conditional Interventional Distributions.’ UAI2006 and ‘Complete Identification Methods for the Causal Hierarchy’ JMLR 2008

**Parameters**

- **cm** (CausalModel (page 236)) – the causal model
- **Y** (Set[str]) – The variables of interest (named following the paper)
- **X** (Set[str]) – The variable of intervention (named following the paper)
- **P** (Optional[ASTtree (page 241)]) – The ASTtree representing the calculus in construction

**Return type**

ASTtree (page 241)

**Returns**

the ASTtree representing the calculus

## 1.9.4 Other functions

pyAgrum.causal.backdoor\_generator(*bn, cause, effect, not\_bd=None*)

Generates backdoor sets for the pair of nodes (*cause, effect*) in the graph *bn* excluding the nodes in the set *not\_bd* (optional)

**Parameters**

- **bn** (pyAgrum.BayesNet (page 64)) –
- **cause** (int) –

- **effect** (*int*) –
- **not\_bd** (*Set[int]* *default=None*) –

#### Yields

*List[int]* – the different backdoors

`pyAgrum.causal.frontdoor_generator(bn, x, y, not_fd=None)`

Generates frontdoor sets for the pair of nodes (*x*, *y*) in the graph *bn* excluding the nodes in the set *not\_fd* (optional)

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) –
- **x** (*int*) –
- **y** (*int*) –
- **not\_fd** (*Set[int]* *default=None*) –

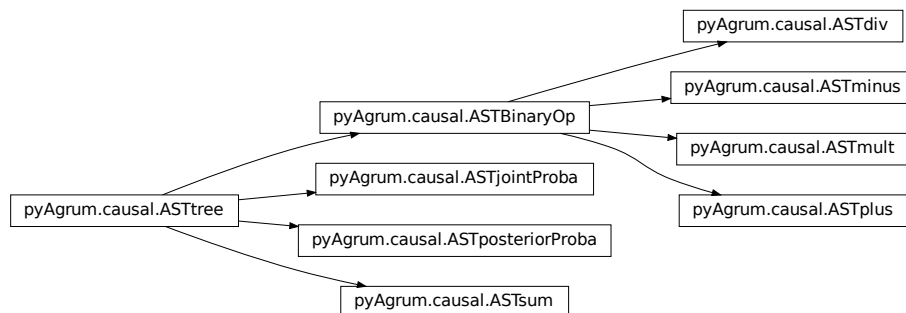
#### Yields

*List[int]* – the different frontdoors

## 1.9.5 Abstract Syntax Tree for Do-Calculus

The pyCausal package compute every causal query into an Abstract Syntax Tree (`CausalFormula`) that represents the exact computations to be done in order to answer to the probabilistic causal query.

The different types of node in an `CausalFormula` are presented below and are organized as a hierarchy of classes from `pyAgrum.causal.ASTtree` (page 241).



### Internal node structure

**class** `pyAgrum.causal.ASTtree`(*typ*, *verbose=False*)

Represents a generic node for the `CausalFormula`. The type of the node will be registered in a string.

#### Parameters

- **typ** (*str*) – the type of the node (will be specified in concrete children classes).
- **verbose** (*bool*) – if True, add some messages

#### `copy()`

Copy an `CausalFormula` tree

**Returns**

the new causal tree

**Return type**

[ASTtree](#) (page 241)

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** ([pyAgrum.BayesNet](#) (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type**

[pyAgrum.Potential](#) (page 53)

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str****Returns**

the type of the node

**Return type**

str

**class** `pyAgrum.causal.ASTBinaryOp`(*typ, op1, op2*)

Represents a generic binary node for the CausalFormula. The op1 and op2 are the two operands of the class.

**Parameters**

- **typ** (*str*) – the type of the node (will be specified in concrete children classes)
- **op1** ([ASTtree](#) (page 241)) – left operand
- **op2** ([ASTtree](#) (page 241)) – right operand

**copy()**

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**

[\*ASTtree\*](#) (page 241)

**eval(contextual\_bn)**

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** ([`pyAgrum.BayesNet`](#) (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type**

[\*pyAgrum.Potential\*](#) (page 53)

**fastToLatex(nameOccur)**

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property op1: [\*ASTtree\*](#) (page 241)****Returns**

the left operand

**Return type**

[\*ASTtree\*](#) (page 241)

**property op2: [\*ASTtree\*](#) (page 241)****Returns**

the right operand

**Return type**

[\*ASTtree\*](#) (page 241)

**protectToLatex(nameOccur)**

Create a protected LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex(nameOccur=None)**

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str****Returns**

the type of the node

**Return type**

str

**Basic Binary Operations****class** pyAgrum.causal.ASTplus(*op1*, *op2*)

Represents the sum of 2 causal.ASTtree

**Parameters**

- **op1** ([ASTtree](#) (page 241)) – left operand
- **op2** ([ASTtree](#) (page 241)) – right operand

**copy()**

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**[ASTtree](#) (page 241)**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters****contextual\_bn** ([pyAgrum.BayesNet](#) (page 64)) – the observational Bayesian network in which will be done the computations**Returns**

the resulting Potential

**Return type**[pyAgrum.Potential](#) (page 53)**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

LaTeX representation of the tree

**Return type**

str

**property** op1: [ASTtree](#) (page 241)**Returns**

the left operand

**Return type**[ASTtree](#) (page 241)**property** op2: [ASTtree](#) (page 241)**Returns**

the right operand

**Return type**[ASTtree](#) (page 241)**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str



**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str**

**Returns**

the type of the node

**Return type**

str

**class** pyAgrum.causal.ASTminus(*op1, op2*)

Represents the subtraction of 2 causal.ASTtree

**Parameters**

- **op1** ([ASTtree](#) (page 241)) – left operand
- **op2** ([ASTtree](#) (page 241)) – right operand

**copy**()

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**

[ASTtree](#) (page 241)

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** ([pyAgrum.BayesNet](#) (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type**

[pyAgrum.Potential](#) (page 53)

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property op1: [ASTtree](#) (page 241)**

**Returns**

the left operand

**Return type**

[ASTtree](#) (page 241)

**property op2: [ASTtree](#) (page 241)**

**Returns**

the right operand

**Return type***ASTtree* (page 241)**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters****nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str****Returns**

the type of the node

**Return type**

str

**class** pyAgrum.causal.ASTdiv(*op1, op2*)

Represents the division of 2 causal.ASTtree

**Parameters**

- **op1** (*ASTtree* (page 241)) – left operand
- **op2** (*ASTtree* (page 241)) – right operand

**copy**()

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type***ASTtree* (page 241)**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters****contextual\_bn** (*pyAgrum.BayesNet* (page 64)) – the observational Bayesian network in which will be done the computations**Returns**

the resulting Potential

**Return type***pyAgrum.Potential* (page 53)**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

LaTeX representation of the tree

**Return type**

str

**property op1:** [ASTtree](#) (page 241)

**Returns**

the left operand

**Return type**

[ASTtree](#) (page 241)

**property op2:** [ASTtree](#) (page 241)

**Returns**

the right operand

**Return type**

[ASTtree](#) (page 241)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type:** str

**Returns**

the type of the node

**Return type**

str

**class** pyAgrum.causal.**ASTmult**(*op1, op2*)

Represents the multiplication of 2 causal.ASTtree

**Parameters**

- **op1** ([ASTtree](#) (page 241)) – left operand
- **op2** ([ASTtree](#) (page 241)) – right operand

**copy**()

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**

[ASTtree](#) (page 241)

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** ([pyAgrum.BayesNet](#) (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type***pyAgrum.Potential* (page 53)**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

LaTeX representation of the tree

**Return type**

str

**property op1:** *ASTtree* (page 241)**Returns**

the left operand

**Return type***ASTtree* (page 241)**property op2:** *ASTtree* (page 241)**Returns**

the right operand

**Return type***ASTtree* (page 241)**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters****nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters****nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable**Returns**

LaTeX representation of the tree

**Return type**

str

**property type:** str**Returns**

the type of the node

**Return type**

str

## Complex operations

**class** `pyAgrum.causal.ASTsum`(*var*, *term*)

Represents a sum over a variable of a `causal.ASTtree`.

### Parameters

- **var** (*str*) – name of the variable on which to sum
- **term** (`ASTtree` (page 241)) – the tree to be evaluated

**copy**()

Copy an CausalFormula tree

### Returns

the new causal tree

### Return type

`ASTtree` (page 241)

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

### Parameters

**contextual\_bn** (`pyAgrum.BayesNet` (page 64)) – the observational Bayesian network in which will be done the computations

### Returns

the resulting Potential

### Return type

`pyAgrum.Potential` (page 53)

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

### Parameters

**nameOccur** (`Dict[str, int]`) – the number of occurrence for each variable

### Returns

LaTeX representation of the tree

### Return type

str

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

### Parameters

**nameOccur** (`Dict[str, int]`) – the number of occurrence for each variable

### Returns

a protected version of LaTeX representation of the tree

### Return type

str

**property term**: `ASTtree` (page 241)

### Returns

the term to sum

### Return type

`ASTtree` (page 241)

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

### Parameters

**nameOccur** (`Dict[str, int]` *default=None*) – the number of occurrence for each variable

### Returns

LaTeX representation of the tree

### Return type

str

**property type: str**

**Returns**

the type of the node

**Return type**

str

**class** pyAgrum.causal.ASTjointProba(*varNames*)

Represent a joint probability in the base observational part of the causal.CausalModel

**Parameters**

**varNames** (*Set[str]*) – a set of variable names

**copy()**

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**

*ASTtree* (page 241)

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** (*pyAgrum.BayesNet* (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type**

*pyAgrum.Potential* (page 53)

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]* *default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str**

**Returns**

the type of the node

**Return type**

str

**property** varNames: Set[str]**Returns**

the set of names of var

**Return type**

Set[str]

**class** pyAgrum.causal.ASTposteriorProba(*bn, varset, knw*)

Represent a conditional probability  $P_{bn}(vars|knw)$  that can be computed by an inference in a BN.

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 64)) – the pyAgrum:pyAgrum.BayesNet
- **varset** (Set[str]) – a set of variable names (in the BN) conditioned in the posterior
- **knw** (Set[str]) – a set of variable names (in the BN) conditioning in the posterior

**property** bn: [BayesNet](#) (page 64)**Returns**the observationnal BayesNet in  $P_{bn}(vars|knw)$ **Return type**[pyAgrum.BayesNet](#) (page 64)**copy()**

Copy an CausalFormula tree

**Returns**

the new causal tree

**Return type**[ASTree](#) (page 241)**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN

**Parameters**

**contextual\_bn** ([pyAgrum.BayesNet](#) (page 64)) – the observational Bayesian network in which will be done the computations

**Returns**

the resulting Potential

**Return type**[pyAgrum.Potential](#) (page 53)**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of the ASTree

**Parameters**

**nameOccur** (Dict[str, int]) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property** knw: Set[str]**Returns**(Conditioning) knw in  $P_{bn}(vars|knw)$ **Return type**

Set[str]

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int]*) – the number of occurrence for each variable

**Returns**

a protected version of LaTeX representation of the tree

**Return type**

str

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Parameters**

**nameOccur** (*Dict[str, int] default=None*) – the number of occurrence for each variable

**Returns**

LaTeX representation of the tree

**Return type**

str

**property type: str****Returns**

the type of the node

**Return type**

str

**property vars: Set[str]****Returns**

(Conditioned) vars in  $P_{bn}(vars|knw)$

**Return type**

Set[str]

## 1.9.6 Exceptions

**class** pyAgrum.causal.**HedgeException**(*msg, observables, gs*)

Represents an hedge exception for a causal query

**Parameters**

- **msg** (*str*) –
- **observables** (*NameSet*) –
- **gs** –

**args****class** pyAgrum.causal.**UnidentifiableException**(*msg*)

Represents an unidentifiability for a causal query

**Parameters**

**msg** (*str*) –

**args**



### 1.9.7 Notebook's tools for causality

This file defines some helpers for handling causal concepts in notebooks

`pyAgrum.causal.notebook.getCausalImpact(model, on, doing, knowing=None, values=None)`

return a HTML representing of the three values defining a causal impact : formula, value, explanation

#### Parameters

- **model** ([CausalModel](#) (page 236)) – the causal model
- **on** (*str* | *Set[str]*) – the impacted variable(s)
- **doing** (*str* | *Set[str]*) – the interventions
- **knowing** (*str* | *Set[str]*) – the observations
- **values** (*Dict[str,int]* *default=None*) – value for certain variables

#### Return type

HTML

`pyAgrum.causal.notebook.getCausalModel(cm, size=None)`

return a HTML representing the causal model

#### Parameters

- **cm** ([CausalModel](#) (page 236)) – the causal model
- **size** (*int* | *str*) – the size of the rendered graph

#### Returns

the dot representation

#### Return type

pydot.Dot

`pyAgrum.causal.notebook.showCausalImpact(model, on, doing, knowing=None, values=None)`

display a HTML representing of the three values defining a causal impact : formula, value, explanation

#### Parameters

- **model** ([CausalModel](#) (page 236)) – the causal model
- **on** (*str* | *Set[str]*) – the impacted variable(s)
- **doing** (*str* | *Set[str]*) – the interventions
- **knowing** (*str* | *Set[str]*) – the observations
- **values** (*Dict[str,int]* *default=None*) – value for certain variables

`pyAgrum.causal.notebook.showCausalModel(cm, size=None)`

Shows a pydot svg representation of the causal DAG

#### Parameters

- **cm** ([CausalModel](#) (page 236)) – the causal model
- **size** (*int* | *str*) – the size of the rendered graph

## 1.10 pyAgrum.skbn documentation

Probabilistic classification in pyAgrum aims to propose a scikit-learn-like (binary and multi-class) classifier class that can be used in the same codes as scikit-learn classifiers. Moreover, even if the classifier wraps a full Bayesian network, skbn optimally encodes the classifier using the smallest set of needed features following the d-separation criterion (Markov Blanket).

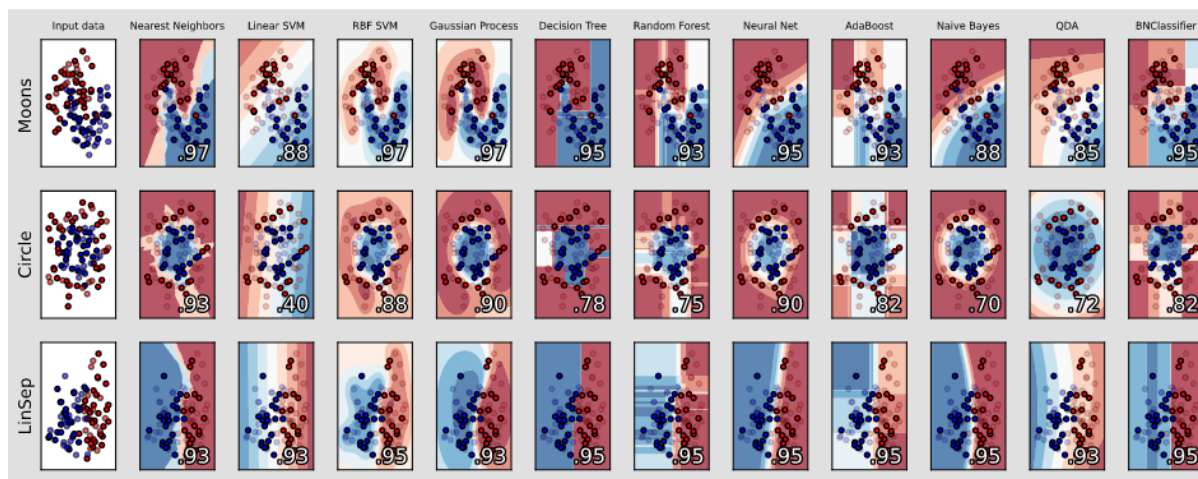


Fig. 1: An example from scikit-learn ([https://scikit-learn.org/stable/auto\\_examples/classification/plot\\_classifier\\_comparison.html](https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html)) where a last column with a BNClassifier has been added flawlessly (see [this notebooks](#)).

The module proposes to wrap the pyAgrum's learning algorithms and some others (naive Bayes, TAN, Chow-Liu tree) in the fit method of a classifier. In order to be used with continuous variable, the module proposes also some different discretization methods.

skbn is a set of pure python3 scripts based on pyAgrum's tools.

### Tutorials

Notebooks on [scikit-learn compliant classifiers in pyAgrum](#),

### Reference

#### 1.10.1 Classifier using Bayesian networks

```
class pyAgrum.skbn.BNClassifier(learningMethod='GHC', prior=None, scoringType='BIC',
                               constraints=None, priorWeight=1,
                               possibleSkeleton=None, DirichletCsv=None,
                               discretizationStrategy='quantile', discretizationNbBins=5,
                               discretizationThreshold=25, usePR=False,
                               significant_digit=10)
```

Represents a (scikit-learn compliant) classifier which uses a BN to classify. A BNClassifier is build using

- a Bayesian network,
- a database and a learning algorithm and parameters
- the use of BNDiscretizer to discretize with different algorithms some variables.

#### parameters:

##### **learningMethod:** str

A string designating which type of learning we want to use. Possible values are: Chow-Liu, NaiveBayes, TAN, MIIC + (MDL ou NML), GHC, 3off2 +

(MDL ou NML), Tabu. GHC designates Greedy Hill Climbing. MIIC designates Multivariate Information based Inductive Causation TAN designates Tree-augmented NaiveBayes Tabu designates Tabu list searching

**prior: str**

A string designating the type of a priorsmoothing we want to use. Possible values are Smoothing, BDeu, Dirichlet and NoPrior . Note: if using Dirichlet smoothing DirichletCsv cannot be set to none By default (when prior is None) : a smoothing(0.01) is applied.

**scoringType: str**

A string designating the type of scoring we want to use. Since scoring is used while constructing the network and not when learning its parameters, the scoring will be ignored if using a learning algorithm with a fixed network structure such as Chow-Liu, TAN or NaiveBayes. possible values are: AIC, BIC, BD, BDeu, K2, Log2 AIC means Akaike information criterion BIC means Bayesian Information criterion BD means Bayesian-Dirichlet scoring BDeu means Bayesian-Dirichlet equivalent uniform Log2 means log2 likelihood ratio test

**constraints: dict()**

A dictionary designating the constraints that we want to put on the structure of the Bayesian network. Ignored if using a learning algorithm where the structure is fixed such as TAN or NaiveBayes. the keys of the dictionary should be the strings "PossibleEdges", "MandatoryArcs" and "ForbiddenArcs". The format of the values should be a tuple of strings (tail,head) which designates the string arc from tail to head. For example if we put the value ("x0"."y") in MandatoryArcs the network will surely have an arc going from x0 to y. Note: PossibleEdge allows between nodes x and y allows for either (x,y) or (y,x) (or none of them) to be added to the Bayesian network, while the others are not symmetric.

**priorWeight: double**

The weight used for a prior.

**possibleSkeleton: pyAgrum.undigraph**

An undirected graph that serves as a possible skeleton for the Bayesian network

**DirichletCsv: str**

the file name of the csv file we want to use for the dirichlet prior. Will be ignored if prior is not set to Dirichlet.

**discretizationStrategy: str**

sets the default method of discretization for this discretizer. This method will be used if the user has not specified another method for that specific variable using the setDiscretizationParameters method possible values are: 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'

**defaultNumberOfBins: str or int**

sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. If the method used is NML, this parameter sets the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discretization method is used, this parameter is ignored.

**discretizationThreshold: int or float**

When using default parameters a variable will be treated as continuous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For instance, if you have entered 0.95, the variable will be treated as continuous only if more than 95% of its values are unique.

**usePR: bool**

indicates if the threshold to choose is Prevision-Recall curve's threshold or ROC's threshold by default. ROC curves should be used when there are roughly equal numbers of observations for each class. Precision-Recall curves should be used when there is a moderate to large class imbalance especially for the target's class.

**significant\_digit:**

number of significant digits when computing probabilities

**XYfromCSV**(*filename*, *with\_labels=True*, *target=None*)

Reads the data from a csv file and separates it into an X matrix and a y column vector.

**Parameters**

- **filename** (*str*) – the name of the csv file
- **with\_labels** (*bool*) – tells us whether the csv includes the labels themselves or their indexes.
- **target** (*str or None*) – The name of the column that will be put in the dataframe y. If target is None, we use the target that is already specified in the classifier

**Returns**

Matrix X containing the data, Column-vector containing the class for each data vector in X

**Return type**

Tuple(pandas.DataFrame, pandas.DataFrame)

**fit**(*X=None*, *y=None*, *data=None*, *targetName=None*, *filename=None*)

**parameters:**

**X: {array-like, sparse matrix} of shape (n\_samples, n\_features)**

training data. Warning: Raises ValueError if either filename or targetname is not None. Raises ValueError if y is None.

**y: array-like of shape (n\_samples)**

Target values. Warning: Raises ValueError if either filename or targetname is not None. Raises ValueError if X is None

**data: Union[str, pandas.DataFrame]**

the source of training data : csv filename or pandas.DataFrame. targetName is mandatory to find the class in this source.

**targetName: str**

specifies the name of the targetVariable in the csv file. Warning: Raises ValueError if either X or y is not None. Raises ValueError if filename is None.

**filename: str**

(deprecated, use data instead) specifies the csv file where the training data and target values are located. Warning: Raises ValueError if either X or y is not None. Raises ValueError if targetName is None

**returns:**

void

Fits the model to the training data provided. The two possible uses of this function are *fit(X,y)* and *fit(data=..., targetName=...)*. Any other combination will raise a ValueError

**fromTrainedModel**(*bn*, *targetAttribute*, *targetModality=""*, *copy=False*, *threshold=0.5*, *variableList=None*)

**parameters:**

**bn: pyagrum.BayesNet**

The Bayesian network we want to use for this classifier

**targetAttribute: str**

the attribute that will be the target in this classifier

**targetModality: str**

If this is a binary classifier we have to specify which modality we are looking at if the target attribute has more than 2 possible values if *!= ""*, a binary classifier is created. if *== ""*, a classifier is created that can be non-binary depending on the number of

modalities for targetAttribute. If binary, the second one is taken as targetModality.

**copy: bool**

Indicates whether we want to put a copy of bn in the classifier, or bn itself.

**threshold: double**

The classification threshold. If the probability that the target modality is true is larger than this threshold we predict that modality

**variableList: list(str)**

A list of strings. variableList[i] is the name of the variable that has the index i. We use this information when calling predict to know which column corresponds to which variable. If this list is set to none, then we use the order in which the variables were added to the network.

**returns:**

void

Creates a BN classifier from an already trained pyAgrum Bayesian network

**get\_params(deep=True)**

Get parameters for this estimator.

**Parameters**

**deep** (*bool*, *default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns**

**params** – Parameter names mapped to their values.

**Return type**

dict

**predict(X, with\_labels=True)**

**parameters:**

**X:** *str*, {*array-like*, *sparse matrix*} of shape (*n\_samples*, *n\_features*) or *str*  
test data, can be either *dataFrame*, *matrix* or name of a *csv* file

**with\_labels: bool**

tells us whether the *csv* includes the labels themselves or their indexes.

**returns:**

**y:** *array-like* of shape (*n\_samples*,)

Predicted classes

Predicts the most likely class for each row of input data, with bn's Markov Blanket

**predict\_proba(X)**

Predicts the probability of classes for each row of input data, with bn's Markov Blanket

**Parameters**

**X** (*str* or {*array-like*, *sparse matrix*} of shape (*n\_samples*, *n\_features*) or *str*) – test data, can be either *dataFrame*, *matrix* or name of a *csv* file

**Returns**

Predicted probability for each classes

**Return type**

*array-like* of shape (*n\_samples*,)

**preparedData(X=None, y=None, data=None, filename=None)**

Given an X and a y (or a data source : *filename* or *pandas.DataFrame*), returns a *pandas.DataFrame* with the prepared (especially discretized) values of the base

**Parameters**

- **X** ({*array-like*, *sparse matrix*} of shape (*n\_samples*, *n\_features*)) – training data. Warning: Raises *ValueError* if either *filename* or *targetname* is not *None*. Raises *ValueError* if *y* is *None*.
- **y** (*array-like* of shape (*n\_samples*)) – Target values. Warning: Raises *ValueError* if either *filename* or *targetname* is not *None*. Raises *ValueError* if *X* is *None*
- **data** (*Union[str, pandas.DataFrame]*) – specifies the *csv* file or the *DataFrame* where the data values are located. Warning: Raises *ValueError*

if either X or y is not None.

- **filename** (*str*) – (deprecated) specifies the csv file where the data are located. Warning: Raises ValueError if either X or y is not None.

**Return type**

pandas.DataFrame

**score**(X, y, sample\_weight=None)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**

- **X** (*array-like of shape (n\_samples, n\_features)*) – Test samples.
- **y** (*array-like of shape (n\_samples,) or (n\_samples, n\_outputs)*) – True labels for X.
- **sample\_weight** (*array-like of shape (n\_samples,)*, *default=None*) – Sample weights.

**Returns**

**score** – Mean accuracy of `self.predict(X)` wrt. y.

**Return type**

float

**set\_params**(\*\*params)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Parameters**

**\*\*params** (*dict*) – Estimator parameters.

**Returns**

**self** – Estimator instance.

**Return type**

estimator instance

**showROC\_PR**(filename, save\_fig=False, show\_progress=False)

Use the `pyAgrum.lib.bn2roc` tools to create ROC and Precision-Recall curve

**Parameters**

- **filename** (*str*) – a csv filename
- **save\_fig** (*bool*) – whether the graph should be saved
- **show\_progress** (*bool*) – indicates if the resulting curve must be printed

## 1.10.2 Discretizer for Bayesian networks

```
class pyAgrum.skbn.BNDiscretizer(defaultDiscretizationMethod='quantile',
                                  defaultNumberOfBins=10, discretizationThreshold=25)
```

Represents a tool to discretize some variables in a database in order to obtain a way to learn a pyAgrum's (discrete) Bayesian networks.

**parameters:**

**defaultDiscretizationMethod: str**

sets the default method of discretization for this discretizer. Possible values are: 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'. This method will be used if the user has not specified another method for that specific variable using the `setDiscretizationParameters` method.

**defaultNumberOfBins: str or int**

sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is

found automatically. If the method used is NML, this parameter sets the the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

**discretizationThreshold: int or float**

When using default parameters a variable will be treated as continous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For example if you have entered 0.95, the variable will be treated as continous only if more than 95% of its values are unique.

**audit**(X, y=None)

**parameters:**

**X:** {array-like, sparse matrix} of shape (n\_samples, n\_features)

training data

**y:** array-like of shape (n\_samples,)

Target values

**returns:**

auditDict: dict()

Audits the passed values of X and y. Tells us which columns in X we think are already discrete and which need to be discretized, as well as the discretization algorithm that will be used to discretize them The parameters which are suggested will be used when creating the variables. To change this the user can manually set discretization parameters for each variable using the setDiscretizationParameters function.

**clear**(clearDiscretizationParameters=False)

**parameters:**

**clearDiscretizationParamaters:** bool

if True, this method also clears the parameters the user has set for each variable and resets them to the default.

**returns:**

void

Sets the number of continous variables and the total number of bins created by this discretizer to 0. If clearDiscretizationParameters is True, also clears the the parameters for discretization the user has set for each variable.

**createVariable**(variableName, X, y=None, possibleValuesY=None)

**parameters:**

**variableName:**

the name of the created variable

**X:** ndarray shape(n,1)

A column vector containing n samples of a feature. The column for which the variable will be created

**y:** ndarray shape(n,1)

A column vector containing the corresponding for each element in X.

**possibleValuesX:** onedimensional ndarray

An ndarray containing all the unique values of X

**possibleValuesY:** onedimensional ndarray

An ndarray containing all the unique values of y

**returnModifiedX:** bool

X could be modified by this function during

**returns:**

**var:** pyagrum.DiscreteVariable

the created variable

Creates a variable for the column passed in as a parameter and places it in the Bayesian network

**discretizationCAIM**(x, y, possibleValuesX, possibleValuesY)

**parametres:**

**x: ndarray with shape (n,1) where n is the number of samples**

Column-vector that contains all the data that needs to be discretized

**y: ndarray with shape (n,1) where n is the number of samples**

Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

**possibleValuesX: one dimensional ndarray**

Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

**possibleValuesY: one dimensional ndarray**

Contains the possible values of y. There should be two possible values since this is a binary classifier

**returns:**

binEdges: a list of the edges of the bins that are chosen by this algorithm

Applies the CAIM algorithm to discretize the values of x

**discretizationElbowMethodRotation**(*discretizationStrategy*, *X*)

**parameters:**

**discretizationStrategy: str**

The method of discretization that will be used. Possible values are: 'quantile' , 'kmeans' and 'uniform'

**X: one dimensional ndarray**

Contains the data that should be discretized

**returns:**

binEdges: the edges of the bins the algorithm has chosen.

Calculates the sum of squared errors as a function of the number of clusters using the discretization strategy that is passed as a parameter. Returns the bins that are optimal for minimizing the variation and the number of bins at the same time. Uses the elbow method to find this optimal point. To find the "elbow" we rotate the curve and look for its minimum.

**discretizationMDLP**(*x*, *y*, *possibleValuesX*, *possibleValuesY*)

**parametres:**

**x: ndarray with shape (n,1) where n is the number of samples**

Column-vector that contains all the data that needs to be discretized

**y: ndarray with shape (n,1) where n is the number of samples**

Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

**possibleValuesX: one dimensional ndarray**

Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

**possibleValuesY: one dimensional ndarray**

Contains the possible values of y. There should be two possible values since this is a binary classifier

**returns:**

binEdges: a list of the edges of the bins that are chosen by this algorithm

Uses the MDLP algorithm described in Fayyad, 1995 to discretize the values of x.

**discretizationNML**(*X*, *possibleValuesX*, *kMax=10*, *epsilon=None*)

**parameters:**

**X: one dimensional ndarray**

array that that contains all the data that needs to be discretized

**possibleValuesX: one dimensional ndarray**

Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside.

**kMax: int**

the maximum number of bins before the algorithm stops itself.

**epsilon: float or None**

the value of epsilon used in the algorithm. Should be as small as possible. If None is passed the value is automatically calculated.

**returns:**



**binEdges**: a list of the edges of the bins that are chosen by this algorithm  
 Uses the discretization algorithm described in “MDL Histogram Density Estimator”, Kontkaken and Myllymaki, 2007 to discretize.

**setDiscretizationParameters**(*variableName=None, method=None, numberOfBins=None*)

**parameters:**

**variableName: str**

the name of the variable you want to set the discretization paramaters of. Set to None to set the new default for this BNClassifier.

**method: str**

The method of discretization used for this variable. Type “NoDiscretization” if you do not want to discretize this variable. Possible values are: ‘NoDiscretization’, ‘quantile’, ‘uniform’, ‘kmeans’, ‘NML’, ‘CAIM’ and ‘MDLP’

**numberOfBins:**

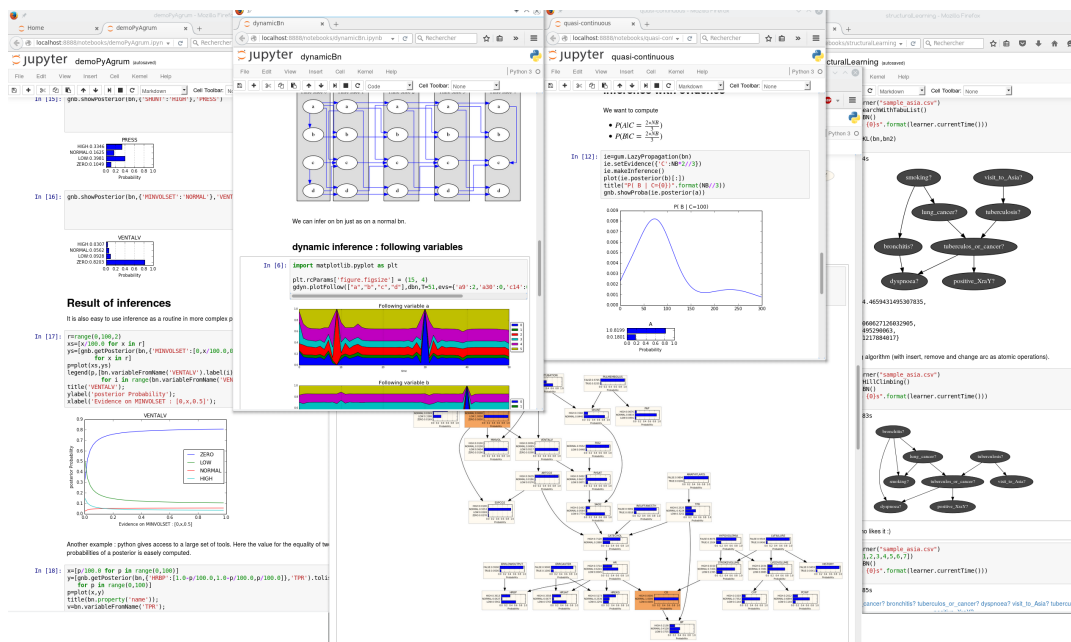
sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string ‘elbowMethod’ so that the best number of bins is found automatically. if the method used is NML, this parameter sets the the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

**returns:**

void

## 1.11 pyAgrum.lib.notebook

`pyAgrum.lib.notebook` aims to facilitate the use of pyAgrum with jupyter notebook (or lab).

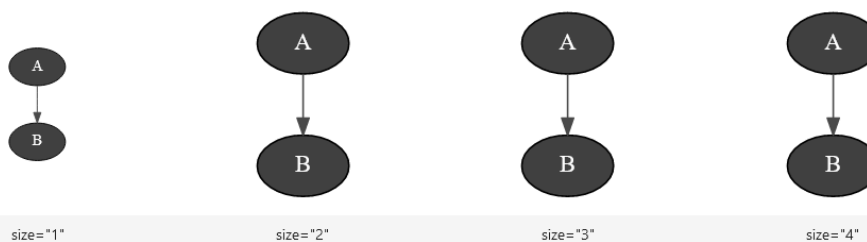


### 1.11.1 Visualization of graphical models

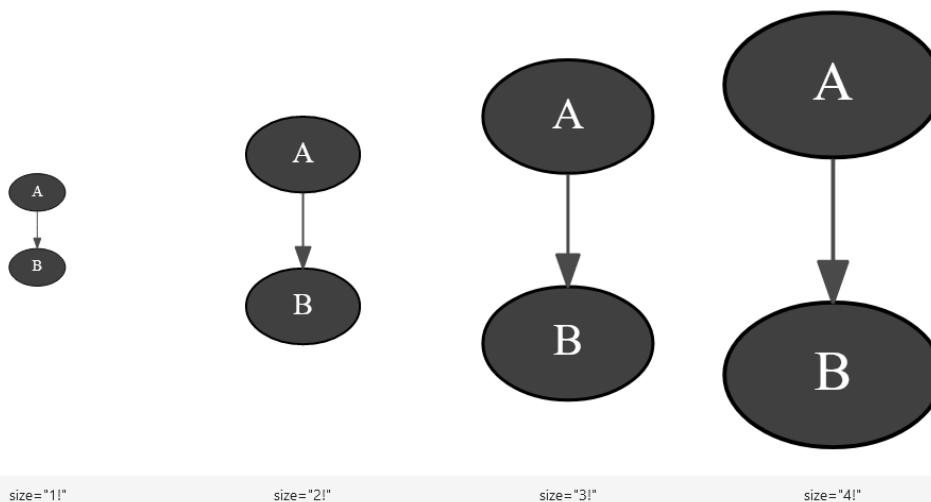
**Important:** For many graphical representations functions, the parameter *size* is directly transferred to *graphviz*. Hence, Its format is a string containing an int. However if *size* ends in an exclamation point “!” (such as *size*=“4!”), then *size* is taken to be the desired minimum size. In this case, if both dimensions of the drawing are less than size, the drawing is scaled up uniformly until at least one dimension equals its dimension in size.

```
1 bn=gum.fastBN("A->B")
2 print("** without '!')
3 gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}") for i in range(1,5)],captions=[f'size="{i}"' for i in range(1,5)])
4
5 print("** with '!')
6 gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}!") for i in range(1,5)],captions=[f'size="{i}!"' for i in range(1,5)])
```

\* without '!'



\* with '!'



```
pyAgrum.lib.notebook.showBN(bn, size=None, nodeColor=None, arcWidth=None,
                             arcColor=None, cmap=None, cmapArc=None)
```

show a Bayesian network

#### Parameters

- **bn** – the Bayesian network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

**Returns**

the graph

```
pyAgrum.lib.notebook.getBN(bn, size=None, nodeColor=None, arcWidth=None,  
                           arcColor=None, cmap=None, cmapArc=None)
```

get a HTML string for a Bayesian network

**Parameters**

- **bn** – the Bayesian network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

**Returns**

the graph

```
pyAgrum.lib.notebook.showInfluenceDiagram(diag, size=None)
```

show an influence diagram as a graph

**Parameters**

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns**

the representation of the influence diagram

```
pyAgrum.lib.notebook.getInfluenceDiagram(diag, size=None)
```

get a HTML string for an influence diagram as a graph

**Parameters**

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns**

the HTML representation of the influence diagram

```
pyAgrum.lib.notebook.showMN(mn, view=None, size=None, nodeColor=None,  
                           factorColor=None, edgeWidth=None, edgeColor=None,  
                           cmap=None, cmapEdge=None)
```

show a Markov network

**Parameters**

- **mn** – the markov network
- **view** – 'graph' | 'factorgraph' | None (default)
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a function returning a value (between 0 and 1) to be shown as a color of factor. (used when view='factorgraph')

- **edgeWidth** – a edgeMap of values to be shown as width of edges (used when view='graph')
- **edgeColor** – a edgeMap of values (between 0 and 1) to be shown as color of edges (used when view='graph')
- **cmap** – color map to show the colors
- **cmapEdge** – color map to show the edge color if distinction is needed

**Returns**

the graph

```
pyAgrum.lib.notebook.getMN(mn, view=None, size=None, nodeColor=None,  
                             factorColor=None, edgeWidth=None, edgeColor=None,  
                             cmap=None, cmapEdge=None)
```

get an HTML string for a Markov network

**Parameters**

- **mn** – the markov network
- **view** – 'graph' | 'factorgraph' | None (default)
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a function returning a value (between 0 and 1) to be shown as a color of factor. (used when view='factorgraph')
- **edgeWidth** – a edgeMap of values to be shown as width of edges (used when view='graph')
- **edgeColor** – a edgeMap of values (between 0 and 1) to be shown as color of edges (used when view='graph')
- **cmap** – color map to show the colors
- **cmapEdge** – color map to show the edge color if distinction is needed

**Returns**

the graph

```
pyAgrum.lib.notebook.showCN(cn, size=None, nodeColor=None, arcWidth=None,  
                             arcColor=None, cmap=None, cmapArc=None)
```

show a credal network

**Parameters**

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

**Returns**

the graph

`pyAgrum.lib.notebook.getCN(cn, size=None, nodeColor=None, arcWidth=None,  
arcColor=None, cmap=None, cmapArc=None)`

get a HTML string for a credal network

#### Parameters

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

#### Returns

the graph

`pyAgrum.lib.notebook.showInference(model, **kwargs)`

show pydot graph for an inference in a notebook

#### Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

#### Returns

the desired representation of the inference

`pyAgrum.lib.notebook.getInference(model, **kwargs)`

get a HTML string for an inference in a notebook

#### Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

**Returns**

the desired representation of the inference

pyAgrum.lib.notebook.**showJunctionTree**(bn, withNames=True, size=None)

Show a junction tree of a Bayesian network

**Parameters**

- **bn** (pyAgrum.BayesNet (page 64)) – the model
- **withNames** (*bool*) – names or id in the graph (names can created very large nodes)
- **size** (*float / str*) – size of the rendered graph

pyAgrum.lib.notebook.**getJunctionTree**(bn, withNames=True, size=None)

get a HTML string for a junction tree (more specifically a join tree)

**Parameters**

- **bn** – the Bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

**Returns**

the HTML representation of the graph

### 1.11.2 Visualization of Potentials

`pyAgrum.lib.notebook.showProba(p, scale=1.0)`

Show a mono-dim Potential (a marginal)

#### Parameters

- **p** (`pyAgrum.Potential` (page 53)) – the marginal to show
- **scale** (`float`) – the zoom factor

`pyAgrum.lib.notebook.getPosterior(bn, evs, target)`

shortcut for `proba2histo(gum.getPosterior(bn, evs, target))`

#### Parameters

- **bn** (`gum.BayesNet`) – the BayesNet
- **evs** (`dict(str->int)`) – map of evidence
- **target** (`str`) – name of target variable

#### Returns

the matplotlib graph

`pyAgrum.lib.notebook.showPosterior(bn, evs, target)`

shortcut for `showProba(gum.getPosterior(bn, evs, target))`

#### Parameters

- **bn** – the BayesNet
- **evs** – map of evidence
- **target** – name of target variable

`pyAgrum.lib.notebook.getPotential(pot, digits=None, withColors=None, varnames=None)`

return a HTML string of a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

#### Parameters

- **pot** (`gum.Potential`) – the potential to show
- **digits** (`int`) – number of digits to show
- **withColors** (`bool`) – bgcolor for proba cells or not
- **varnames** (`List[str]`) – the aliases for variables name in the table

#### Returns

the html representation of the Potential (as a string)

#### Return type

str

`pyAgrum.lib.notebook.showPotential(pot, digits=None, withColors=None, varnames=None)`

show a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

#### Parameters

- **pot** (`gum.Potential`) – the potential to show
- **digits** (`int`) – number of digits to show
- **withColors** (`bool`) – bgcolor for proba cells or not
- **varnames** (`List[str]`) – the aliases for variables name in the table

### 1.11.3 Visualization of graphs

`pyAgrum.lib.notebook.getDot(dotstring, size=None)`

get an HTML representation of a dot string

**Parameters**

- **dotstring** (*str*) – the dot string
- **size** (*float / str*) – size of the rendered graph

**Return type**

the HTML representation of the dot string

`pyAgrum.lib.notebook.showDot(dotstring, size=None)`

show a dot string as a graph

**Parameters**

- **dotstring** (*str*) – the dot string
- **size** (*float / str*) – size of the rendered graph

`pyAgrum.lib.notebook.getGraph(gr, size=None)`

get an HTML representation of a pydot graph

**Parameters**

- **gr** (*pydot.Dot*) – the graph
- **size** (*float / str*) – the size of the rendered graph

**Return type**

the HTML representation of the graph (as a string)

`pyAgrum.lib.notebook.showGraph(gr, size=None)`

show a pydot graph in a notebook

**Parameters**

- **gr** (*pydot.Dot*) – the graph
- **size** (*float / str*) – the size of the rendered graph

### 1.11.4 Visualization of approximation algorithm

`pyAgrum.lib.notebook.animApproximationScheme(apsc, scale=<ufunc 'log10'>)`

show an animated version of an approximation algorithm

**Parameters**

- **apsc** – the approximation algorithm
- **scale** – a function to apply to the figure



### 1.11.5 Helpers

`pyAgrum.lib.notebook.configuration()`

Display the collection of dependance and versions

`pyAgrum.lib.notebook.sideBySide(*args, **kwargs)`

display side by side args as HTML fragment (using string, `_repr_html_()` or `str()`)

#### Parameters

- **args** – HTML fragments as string arg, `arg._repr_html_()` or `str(arg)`
- **captions** – list of strings (captions)

## 1.12 pyAgrum.lib.image

`pyAgrum.lib.image` aims to graphically export models and inference using `pydot` (<https://pypi.org/project/pydot/>) (and then `graphviz` (<https://graphviz.org/>)).

For more details, see *this notebook* (page 483).

```

1 import pyAgrum as gum
2 from pyAgrum.lib.image as gumimage
3
4 bn = gum.fastBN("a->b->d;a->c->d[3]->e;f->b")
5 gumimage.export(bn,"out/test_export.png",
6                 nodeColor={'a': 1,
7                             'b': 0.3,
8                             'c': 0.4,
9                             'd': 0.1,
10                            'e': 0.2,
11                            'f': 0.5},
12                 arcColor={(0, 1): 0.2,
13                           (1, 2): 0.5},
14                 arcWidth={(0, 3): 0.4,
15                           (3, 2): 0.5,
16                           (2,4) :0.6})

```

### 1.12.1 Visualization of models and inference

`pyAgrum.lib.image.export(model, filename=None, **kwargs)`

export the graphical representation of the model in filename (png, pdf,etc.)

#### Parameters

- **model** (`pyAgrum.GraphicalModel`) – the model to show (`pyAgrum.BayesNet`, `pyAgrum.MarkovNet`, `pyAgrum.InfluenceDiagram` or `pyAgrum.Potential`)
- **filename** (`str`) – the name of the resulting file (suffix in [`'pdf'`, `'png'`, `'fig'`, `'jpg'`, `'svg'`, `'ps'`]). If filename is None, the result is a `np.array` ready to be used with `imshow()`.

---

**Note:** Model can also just possess a method `toDot()` or even be a simple string in dot syntax.

---

`pyAgrum.lib.image.exportInference(model, filename=None, **kwargs)`

the graphical representation of an inference in a notebook

#### Parameters

- **model** (*pyAgrum:GraphicalModel*) – the model in which to infer (*pyAgrum.BayesNet*, *pyAgrum.MarkovNet* or *pyAgrum.InfluenceDiagram*)
- **filename** (*str*) – the name of the resulting file (suffix in [`'pdf'`, `'png'`, `'ps'`]). If filename is None, the result is a np.array ready to be used with `imshow()`.
- **engine** (*pyAgrum.Inference*) – inference algorithm used. If None, *gum.LazyPropagation* will be used for *BayesNet*, *gum.ShaferShenoy* for *gum.MarkovNet* and *gum.ShaferShenoyLIMIDInference* for *gum.InfluenceDiagram*.
- **evs** (*Dict[str, str/int]*) – map of evidence
- **targets** (*Set[str/int]*) – set of targets
- **size** (*str*) – size of the rendered graph
- **nodeColor** (*Dict[int, float]*) – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** (*Dict[int, float]*) – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** (*Dict[(int, int), float]*) – a arcMap of values to be shown as width of arcs
- **arcColor** (*Dict[(int, int), float]*) – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** (*matplotlib.colors.ColorMap*) – color map to show the color of nodes and arcs
- **cmapArc** (*matplotlib.colors.ColorMap*) – color map to show the vals of Arcs.
- **graph** (*pyAgrum.Graph*) – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** (*str*) – graph | factorgraph | None (default) for Markov network

**Returns**

the desired representation of the inference

**Return type**

str|dot.Dot

## 1.13 pyAgrum.lib.explain

The purpose of `pyAgrum.lib.explain` is to give tools to explain and interpret the structure and parameters of a Bayesian network.

### 1.13.1 Dealing with independence

`pyAgrum.lib.explain.independenceListForPairs`(*bn, filename, target=None, plot=True, alphabetic=False*)

get the p-values of the chi2 test of a (as simple as possible) independence proposition for every non arc.

**Parameters**

- **bn** (*gum.BayesNet*) – the Bayesian network
- **filename** (*str*) – the name of the csv database

- **alphabetic** (*bool*) – if True, the list is alphabetically sorted else it is sorted by the p-value
- **target** (*(optional) str or int*) – the name or id of the target variable
- **plot** (*bool*) – if True, plot the result

**Return type**  
the list

### 1.13.2 Dealing with mutual information and entropy

```
pyAgrum.lib.explain.getInformation(bn, evs=None, size=None,
                                  cmap=<matplotlib.colors.LinearSegmentedColormap
                                  object>)
```

get a HTML string for a bn annotated with results from inference : entropy and mutual information

#### Parameters

- **bn** ([pyAgrum.BayesNet](#) (page 64)) – the model
- **evs** (*Dict[str/int, str/int/List[float]]*) – the observations
- **size** (*int/str*) – size of the rendered graph
- **cmap** (*matplotlib.colours.Colormap*) – the cmap

#### Returns

return the HTML string

#### Return type

str

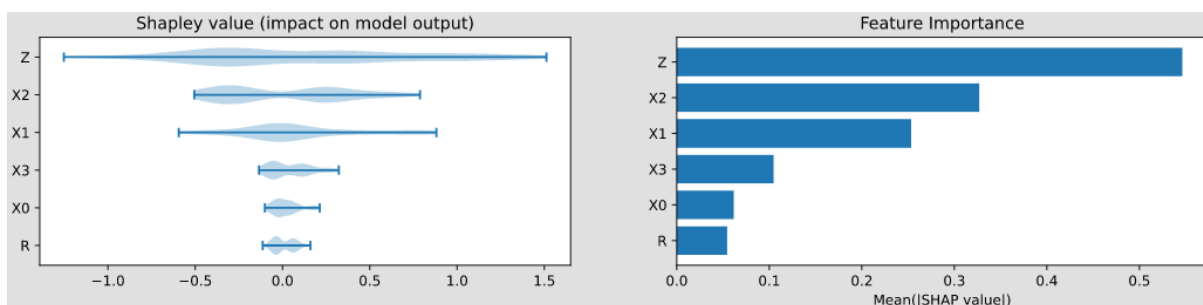
```
pyAgrum.lib.explain.showInformation(bn, evs=None, size=None,
                                    cmap=<matplotlib.colors.LinearSegmentedColormap
                                    object>)
```

display a bn annotated with results from inference : entropy and mutual information

#### Parameters

- **bn** ([pyAgrum.BayesNet](#) (page 64)) – the model
- **evs** (*Dict[str/int, str/int/List[float]]*) – the observations
- **size** (*int/str*) – size of the rendered graph
- **cmap** (*matplotlib.colours.Colormap*) – the cmap

### 1.13.3 Dealing with ShapValues



```
class pyAgrum.lib.explain.ShapValues(bn, target)
```

Bases: object

The ShapValue class implements the calculation of Shap values in Bayesian networks.

The main implementation is based on Conditional Shap values<sup>3</sup>, but the Interventional calculation method proposed in<sup>2</sup> is also present. In addition, a new causal method, based on<sup>1</sup>, is implemented which is well suited for Bayesian networks.

```
causal(train, plot=False, plot_importance=False, percentage=False)
```

Compute the causal Shap Values for each variables.

**Parameters**

- **train** (*pandas.DataFrame*) – the database
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot\_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

**Return type**

a dictionary Dict[str,float]

```
conditional(train, plot=False, plot_importance=False, percentage=False)
```

Compute the conditional Shap Values for each variables.

**Parameters**

- **train** (*pandas.DataFrame*) – the database
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot\_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

**Return type**

a dictionary Dict[str,float]

```
marginal(train, sample_size=200, plot=False, plot_importance=False, percentage=False)
```

Compute the marginal Shap Values for each variables.

**Parameters**

- **train** (*pandas.DataFrame*) – the database
- **sample\_size** (*int*) – The computation of marginal ShapValue is very slow. The parameter allow to compute only on a fragment of the database.
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot\_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

**Return type**

a dictionary Dict[str,float]

```
showShapValues(results, cmap='plasma')
```

**Parameters**

- **results** (*dict[str,float]*) – The (Shap) values associates to each variable
- **cmap** (*Matplotlib.ColorMap*) – The colormap used for colouring the nodes

**Return type**

a pydot.graph

---

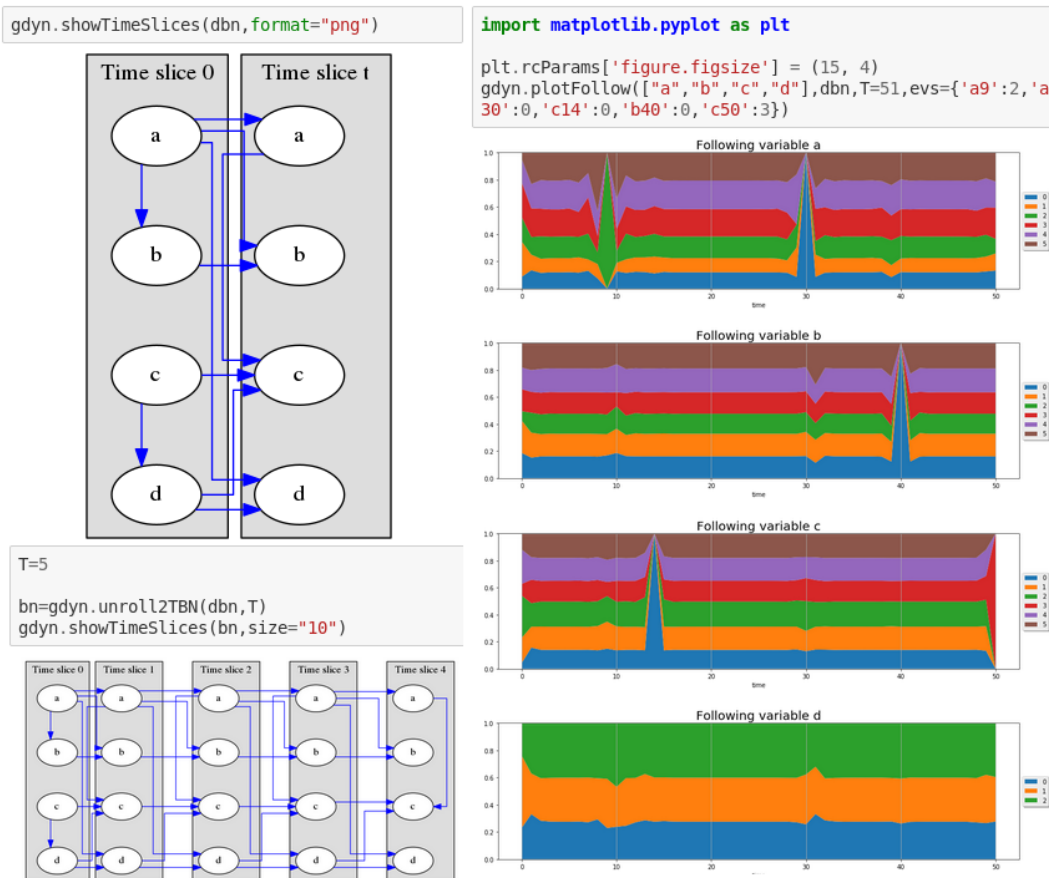
<sup>3</sup> Lundberg, S. M., & Su-In, L. (2017). A Unified Approach to Interpreting Model. 31st Conference on Neural Information Processing Systems. Long Beach, CA, USA.

<sup>2</sup> Janzing, D., Minorics, L., & Blöbaum, P. (2019). Feature relevance quantification in explainable AI: A causality problem. arXiv: Machine Learning. Retrieved 6 24, 2021, from <https://arxiv.org/abs/1910.13413>

<sup>1</sup> Heskens, T., Sijben, E., Bucur, I., & Claassen, T. (2020). Causal Shapley Values: Exploiting Causal Knowledge. 34th Conference on Neural Information Processing Systems. Vancouver, Canada.

## 1.14 pyAgrum.lib.dynamicBN

dynamic Bayesian Network are a special class of BNs where variables can be subscripted by a (discrete) time. See [this notebook](#) (page 367).



The purpose of this module is to provide basic tools for dealing with dynamic Bayesian Network (and inference) : modeling, visualisation, inference.

`pyAgrum.lib.dynamicBN.getTimeSlices(dbn, size=None)`

Try to correctly represent dBN and 2TBN as an HTML string

### Parameters

- **dbn** (`pyAgrum.BayesNet` (page 64)) – a 2TBN or an unrolled BN
- **size** (`int` or `str`) – size of the fig

`pyAgrum.lib.dynamicBN.getTimeSlicesRange(dbn)`

get the range and (name,radical) of each variables

### Parameters

**dbn** (`gum.BayesNet`) – a 2TBN or an unrolled BN

### Returns

all the timeslice of a dbn : ['0','t'] for a classic 2TBN, range(T) for a classic unrolled BN

### Return type

Dict[str,List[T[str,str]]]

`pyAgrum.lib.dynamicBN.is2TBN(bn)`

Check if bn is a 2 TimeSlice Bayesian network

**Parameters**

**bn** ([pyAgrum.BayesNet](#) (page 64)) – the Bayesian network

**Returns**

True if the BN is syntactically correct to be a 2TBN

**Return type**

bool

`pyAgrum.lib.dynamicBN.plotFollow(lovars, twoTdbn, T, evs)`

plots modifications of variables in a 2TDN knowing the size of the time window (T) and the evidence on the sequence.

**Parameters**

- **lovars** – list of variables to follow
- **twoTdbn** – the two-timeslice dbn
- **T** – the time range
- **evs** – observations

`pyAgrum.lib.dynamicBN.plotFollowUnrolled(lovars, dbn, T, evs, vars_title=None)`

plot the dynamic evolution of a list of vars with a dBN

**Parameters**

- **lovars** – list of variables to follow
- **dbn** – the unrolled dbn
- **T** – the time range
- **evs** – observations
- **vars\_title** – string for default or a dictionary with the variable name as key and the respective title as value.

`pyAgrum.lib.dynamicBN.realNameFrom2TBNname(name, ts)`

@return dynamic name from static name and timeslice (no check)

`pyAgrum.lib.dynamicBN.showTimeSlices(dbn, size=None)`

Try to correctly display dBN and 2TBN

**Parameters**

- **dbn** ([pyAgrum.BayesNet](#) (page 64)) – a 2TBN or an unrolled BN
- **size** (*int* or *str*) – size of the fig

`pyAgrum.lib.dynamicBN.unroll2TBN(dbn, nbr)`

unroll a 2TBN given the nbr of timeslices

**Parameters**

- **dbn** ([pyAgrum.BayesNet](#) (page 64)) – a 2TBN or an unrolled BN
- **nbr** (*int*) – the number of timeslice

**Returns**

unrolled BN from a 2TBN and the nbr of timeslices

**Return type**

[pyAgrum.BayesNet](#) (page 64)

## 1.15 other pyAgrum.lib modules

### 1.15.1 bn2roc

The purpose of this module is to provide tools for building ROC and PR from Bayesian Network.

`pyAgrum.lib.bn2roc.showPR(bn, csv_name, target, label, show_progress=True, show_fig=True, save_fig=False, with_labels=True, significant_digits=10)`

Compute the ROC curve and save the result in the folder of the csv file.

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) – a Bayesian network
- **csv\_name** (`str`) – a csv filename
- **target** (`str`) – the target
- **label** (`str`) – the target label
- **show\_progress** (`bool`) – indicates if the progress bar must be printed
- **save\_fig** – save the result ?
- **show\_fig** – plot the results ?
- **with\_labels** – labels in csv ?
- **significant\_digits** – number of significant digits when computing probabilities

`pyAgrum.lib.bn2roc.showROC(bn, csv_name, target, label, show_progress=True, show_fig=True, save_fig=False, with_labels=True, significant_digits=10)`

Compute the ROC curve and save the result in the folder of the csv file.

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) – a Bayesian network
- **csv\_name** (`str`) – a csv filename
- **target** (`str`) – the target
- **label** (`str`) – the target label
- **show\_progress** (`bool`) – indicates if the progress bar must be printed
- **save\_fig** – save the result
- **show\_fig** – plot the results
- **with\_labels** – labels in csv
- **significant\_digits** – number of significant digits when computing probabilities

`pyAgrum.lib.bn2roc.showROC_PR(bn, csv_name, target, label, show_progress=True, show_fig=True, save_fig=False, with_labels=True, show_ROC=True, show_PR=True, significant_digits=10)`

Compute the ROC curve and save the result in the folder of the csv file.

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) – a Bayesian network
- **csv\_name** (`str`) – a csv filename
- **target** (`str`) – the target

- **label** (*str*) – the target label
- **show\_progress** (*bool*) – indicates if the progress bar must be printed
- **save\_fig** – save the result
- **show\_fig** – plot the results
- **with\_labels** – labels in csv
- **show\_ROC** (*bool*) – whether we show the ROC figure
- **show\_PR** (*bool*) – whether we show the PR figure
- **significant\_digits** – number of significant digits when computing probabilities

**Returns**

(pointsROC, thresholdROC, pointsPR, thresholdPR)

**Return type**

tuple

### 1.15.2 bn2scores

The purpose of this module is to provide tools for computing different scores from a BN.

`pyAgrum.lib.bn2scores.checkCompatibility(bn, fields, csv_name)`

check if the variables of the bn are in the fields

**Parameters**

- **bn** (*gum.BayesNet*) – the model
- **fields** (*Dict[str, int]*) – Dict of name, position in the file
- **csv\_name** (*str*) – name of the csv file

**Raises**

**gum.DatabaseError** – if a BN variable is not in fields

**Returns**

return a dictionary of position for BN variables in fields

**Return type**

Dict[int, str]

`pyAgrum.lib.bn2scores.computeScores(bn_name, csv_name, visible=False)`

Compute scores (likelihood, aic, bic, mdl, etc.) from a bn w.r.t to a csv

**Parameters**

- **bn\_name** (`pyAgrum.BayesNet` (page 64) / *str*) – a gum.BayesianNetwork or a filename for a BN
- **csv\_name** (*str*) – a filename for the CSV database
- **visible** (*bool*) – do we show the progress

**Returns**

percentDatabaseUsed, scores

**Return type**

Tuple[float, Dict[str, float]]

`pyAgrum.lib.bn2scores.lines_count(filename)`

count lines in a file



### 1.15.3 bn\_vs\_bn

The purpose of this module is to provide tools for comaring different BNs.

**class** pyAgrum.lib.bn\_vs\_bn.**GraphicalBNComparator**(name1, name2, delta=1e-06)

Bases: object

BNGraphicalComparator allows to compare in multiple way 2 BNs... The smallest assumption is that the names of the variables are the same in the 2 BNs. But some comparisons will have also to check the type and domainSize of the variables. The bns have not exactly the same role : \_bn1 is rather the referent model for the comparison whereas \_bn2 is the compared one to the referent model.

#### Parameters

- **name1** (*str* or [pyAgrum.BayesNet](#) (page 64)) – a BN or a filename for reference
- **name2** (*str* or [pyAgrum.BayesNet](#) (page 64)) – another BN or antoher file-name for comparison

#### dotDiff()

Return a pydot graph that compares the arcs of \_bn1 (reference) with those of self.\_bn2.  
 full black line: the arc is common for both  
 full red line: the arc is common but inverted in \_bn2  
 dotted black line: the arc is added in \_bn2  
 dotted red line: the arc is removed in \_bn2

**Warning:** if pydot is not installed, this function just returns None

#### Returns

the result dot graph or None if pydot can not be imported

#### Return type

pydot.Dot

#### equivalentBNs()

Check if the 2 BNs are equivalent :

- same variables
- same graphical structure
- same parameters

#### Returns

“OK” if bn are the same, a description of the error otherwise

#### Return type

str

#### hamming()

Compute hamming and structural hamming distance

Hamming distance is the difference of edges comparing the 2 skeletons, and Structural Hamming difference is the difference comparing the cpdags, including the arcs' orientation.

#### Returns

A dictionnary containing 'hamming','structural hamming'

#### Return type

dict[double,double]

#### scores()

Compute Precision, Recall, F-score for self.\_bn2 compared to self.\_bn1

precision and recall are computed considering BN1 as the reference

Fscore is  $2 * (\text{recall} * \text{precision}) / (\text{recall} + \text{precision})$  and is the weighted average of Precision and Recall.

dist2opt=square root of  $(1-\text{precision})^2+(1-\text{recall})^2$  and represents the euclidian distance to the ideal point (precision=1, recall=1)

**Returns**

A dictionary containing 'precision', 'recall', 'fscore', 'dist2opt' and so on.

**Return type**

dict[str,double]

**skeletonScores()**

Compute Precision, Recall, F-score for skeletons of self.\_bn2 compared to self.\_bn1

precision and recall are computed considering BN1 as the reference

Fscore is  $2*(\text{recall} * \text{precision})/(\text{recall}+\text{precision})$  and is the weighted average of Precision and Recall.

dist2opt=square root of  $(1-\text{precision})^2+(1-\text{recall})^2$  and represents the euclidian distance to the ideal point (precision=1, recall=1)

**Returns**

A dictionary containing 'precision', 'recall', 'fscore', 'dist2opt' and so on.

**Return type**

dict[str,double]

`pyAgrum.lib.bn_vs_bn.graphDiff(bnref, bncmp, noStyle=False)`

Return a pydot graph that compares the arcs of bnref to bncmp. graphDiff allows bncmp to have less nodes than bnref. (this is not the case in GraphicalBNComparator.dotDiff())

if noStyle is False use 4 styles (fixed in pyAgrum.config) :

- the arc is common for both
- the arc is common but inverted in \_bn2
- the arc is added in \_bn2
- the arc is removed in \_bn2

See graphDiffLegend() to add a legend to the graph. .. warning:: if pydot is not installed, this function just returns None

**Returns**

the result dot graph or None if pydot can not be imported

**Return type**

pydot.Dot

`pyAgrum.lib.bn_vs_bn.graphDiffLegend()`

## 1.16 Functions from pyAgrum

### 1.16.1 Useful functions in pyAgrum

`pyAgrum.about()`

about() for pyAgrum

`pyAgrum.getPosterior(model, evs, target)`

Compute the posterior of a single target (variable) in a BN given evidence

getPosterior uses a VariableElimination inference. If more than one target is needed with the same set of evidence or if the same target is needed with more than one set of evidence, this function is not relevant since it creates a new inference engine every time it is called.

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 64) or [pyAgrum.MarkovNet](#) (page 217)) – The probabilistic Graphical Model
- **evs** (*dictionaryDict*) – {name|id:val, name|id : [ val1, val2 ], ... }
- **target** (*string or int*) – variable name or id

**Return type**

posterior ([pyAgrum.Potential](#) (page 53) or other)

`pyAgrum.generateSample(bn, n=1, name_out=None, show_progress=False, with_labels=True, random_order=True)`

generate a CSV file of samples from a bn.

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 64)) – the Bayes Net from which the sample is generated
- **n** (*int*) – the number of samples
- **name\_out** (*str*) – the name for the output csv filename. If name\_out is None, a pandas.DataFrame is generated
- **show\_progress** (*bool*) – if True, show a progress bar. Default is False
- **with\_labels** (*bool*) – if True, use the labels of the modalities of variables in the csv. If False, use their ids. Default is True
- **random\_order** (*bool*) – if True, the columns in the csv are randomized sorted. Default is True

**Returns**

the log2-likelihood of the generated base or if name\_out is None, the couple (generated pandas.DataFrame, log2-likelihood)

**Return type**

float[Tuple[pandas.DataFrame, float]]

`pyAgrum.generateCSV(bn, name_out, n=1, show_progress=False, with_labels=False, random_order=True)`

Deprecated. Please use `pyAgrum.generateSample` instead.

## 1.16.2 Quick specification of (randomly parameterized) graphical models

aGrUM/pyAgrum offers a so-called “fact” syntax that allows to quickly and compactly specify prototypes of graphical models. These *fastPrototype* aGrUM’s methods have also been wrapped in functions of pyAgrum.

```
gum.fastBN("A[10]->B<-C{top|middle|bottom};B->D")
```

The type of the random variables can be specify with different syntaxes:

- by default, a variable is a [pyAgrum.RangeVariable](#) (page 39) using the default domain size (second argument of the functions).
- with a[10], the variable is a [pyAgrum.RangeVariable](#) (page 39) using 10 as domain size (from 0 to 9)
- with a[3, 7], the variable is a [pyAgrum.RangeVariable](#) (page 39) using a domainSize from 3 to 7
- with a[1, 3.14, 5, 6.2], the variable is a [pyAgrum.DiscretizedVariable](#) (page 31) using the given ticks (at least 3 values)

- with a{top|middle|bottom}, the variable is a *pyAgrum.LabelizedVariable* (page 27) using the given labels (here : 'top', 'middle' and 'bottom').
- with a{-1|5|0|3}, the variable is a *pyAgrum.IntegerVariable* (page 35) using the sorted given values.
- with 'a{-0.5|5.01|0|3.1415}', the variable is a *pyAgrum.NumericalDiscreteVariable* (page 42) using the sorted given values.

---

**Note:**

- If the dot-like string contains such a specification more than once for a variable, the first specification will be used.
  - the CPTs are randomly generated.
- 

`pyAgrum.fastBN(structure, domain_size=2)`

**Create a Bayesian network with a dot-like syntax which specifies:**

- the structure 'a->b->c;b->d<-e;',
- the type of the variables with different syntax (cf documentation).

**Examples**

```
>>> import pyAgrum as gum
>>> bn=gum.fastBN('A->B[1,3]<-C{yes|No}->D[2,4]<-E[1,2.5,3.9]',6)
```

**Parameters**

- **structure** (*str*) – the string containing the specification
- **domain\_size** (*int*) – the default domain size for variables

**Returns**

the resulting bayesian network

**Return type**

*pyAgrum.BayesNet* (page 64)

`pyAgrum.fastMN(structure, domain_size=2)`

**Create a Markov network with a modified dot-like syntax which specifies:**

- the structure 'a-b-c;b-d;c-e;' where each chain 'a-b-c' specifies a factor,
- the type of the variables with different syntax (cf documentation).

**Examples**

```
>>> import pyAgrum as gum
>>> bn=gum.fastMN('A--B[1,3]--C{yes|No};C--D[2,4]--E[1,2.5,3.9]',6)
```

**Parameters**

- **structure** (*str*) – the string containing the specification
- **domain\_size** (*int*) – the default domain size for variables

**Returns**

the resulting Markov network

**Return type***pyAgrum.MarkovNet* (page 217)`pyAgrum.fastID(structure, domain_size=2)`**Create an Influence Diagram with a modified dot-like syntax which specifies:**

- the structure and the type of the variables following *fast syntax* (page 279),
- a prefix for the type of node (chance/decision/utility nodes):
  - `a` : a chance node named 'a' (by default)
  - `$a` : a utility node named 'a'
  - `*a` : a decision node named 'a'

**Examples**

```
>>> import pyAgrum as gum
>>> bn=gum.fastID('A->B[1,3]<-*C{yes|No}->$D<-E[1,2.5,3.9]',6)
```

**Parameters**

- **structure** (*str*) – the string containing the specification
- **domain\_size** (*int*) – the default domain size for variables

**Returns**

the resulting Influence Diagram

**Return type***pyAgrum.InfluenceDiagram* (page 190)

### 1.16.3 Input/Output for Bayesian networks

`pyAgrum.availableBNExts()`

Give the list of all formats known by pyAgrum to save a Bayesian network.

**Returns**

a string which lists all suffixes for supported BN file formats.

`pyAgrum.loadBN(filename, listeners=None, verbose=False, **opts)`

load a BN from a file with optional listeners and arguments

**Parameters**

- **filename** (*str*) – the name of the input file
- **listeners** (*List[object]*) – list of functions to execute when listening
- **verbose** (*bool*) – whether to print or not warning messages
- **system** (*str*) – (for O3PRM) name of the system to flatten in a BN
- **classpath** (*List[str]*) – (for O3PRM) list of folders containing classes

**Returns**

a BN from a file using one of the availableBNExts() suffixes.

**Return type***pyAgrum.BayesNet* (page 64)

## Notes

Listeners could be added in order to monitor its loading.

## Examples

```
>>> import pyAgrum as gum
>>>
>>> # creating listeners
>>> def foo_listener(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end=' ',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadBN('./bn.bif',listeners=[foo_listener,bar_listener])
>>> # .....#.....#.....#.....#..50%
>>> # .....#.....#.....#.....#.....#.....% | bn_
↪loaded
```

`pyAgrum.saveBN(bn, filename)`

save a BN into a file using the format corresponding to one of the available `WriteBNExts()` suffixes.

### Parameters

- **bn** (`pyAgrum.BayesNet` (page 64)) – the BN to save
- **filename** (`str`) – the name of the output file

## 1.16.4 Input/Output for Markov networks

`pyAgrum.availableMNExts()`

Give the list of all formats known by pyAgrum to save a Markov network.

### Returns

a string which lists all suffixes for supported MN file formats.

### Return type

`str`

`pyAgrum.loadMN(filename, listeners=None, verbose=False)`

load a MN from a file with optional listeners and arguments

### Parameters

- **filename** (`str`) – the name of the input file
- **listeners** (`List[Object]`) – list of functions to execute

- **verbose** (*bool*) – whether to print or not warning messages

#### Returns

- *pyAgrum.MarkovNet* – a MN from a file using one of the availableMNExts() suffixes.
- *Listeners could be added in order to monitor its loading.*

#### Examples

```
>>> import pyAgrum as gum
>>>
>>> # creating listeners
>>> def foo_listener(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end=' ',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadMN('./bn.uai',listeners=[foo_listener,bar_listener])
>>> # .....#.....#.....#.....#..50%
>>> # .....#.....#.....#.....#.....#.....% | bn_
↪loaded
```

**pyAgrum.saveMN**(*mn*, *filename*)

save a MN into a file using the format corresponding to one of the availableWriteMNExts() suffixes.

#### Parameters

- **mn** (*pyAgrum.MarkovNet* (page 217)) – the MN to save
- **filename** (*str*) – the name of the output file

### 1.16.5 Input for influence diagram

**pyAgrum.availableIDExt**s()

Give the list of all formats known by pyAgrum to save a influence diagram.

#### Returns

a string which lists all suffixes for supported ID file formats.

#### Return type

str

**pyAgrum.loadID**(*filename*)

read a gum.InfluenceDiagram from a ID file

**Parameters**

**filename** (*str*) – the name of the input file

**Returns**

the InfluenceDiagram

**Return type**

*pyAgrum.InfluenceDiagram* (page 190)

`pyAgrum.saveID(infdiag, filename)`

save an ID into a file using the format corresponding to one of the available `WriteIDExts()` suffixes.

**Parameters**

- **infdiag** (*pyAgrum.InfluenceDiagram* (page 190)) – the Influence Diagram to save
- **filename** (*str*) – the name of the output file

## 1.17 Other functions from aGrUM

### 1.17.1 Listeners

aGrUM includes a mechanism for listening to actions (close to QT signal/slot). Some of them have been ported to pyAgrum :

#### LoadListener

Listeners could be added in order to monitor the progress when loading a `pyAgrum.BayesNet`

```
>>> import pyAgrum as gum
>>>
>>> # creating a new listeners
>>> def foo(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end=' ',flush=True)
>>>
>>> def bar(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>>
>>> gum.loadBN('./bn.bif',listeners=[foo,bar])
>>> # .....#.....#.....#.....#...50%
>>> # .....#.....#.....#.....#.....#.....% | bn loaded
```



## StructuralListener

Listeners could also be added when structural modification are made in a `pyAgrum.BayesNet`:

```
>>> import pyAgrum as gum
>>>
>>> ## creating a BayesNet
>>> bn=gum.BayesNet()
>>>
>>> ## adding structural listeners
>>> bn.addStructureListener(whenNodeAdded=lambda n,s:print(f'adding {n}:{s}
→'),
>>>                               whenArcAdded=lambda i,j: print(f'adding {i}->{j}
→'),
>>>                               whenNodeDeleted=lambda n:print(f'deleting {n}'),
>>>                               whenArcDeleted=lambda i,j: print(f'deleting {i}-
→>{j}'))
>>>
>>> ## adding another listener for when a node is deleted
>>> bn.addStructureListener(whenNodeDeleted=lambda n: print('yes, really_
→deleting '+str(n)))
>>>
>>> ## adding nodes to the BN
>>> l=[bn.add(item,3) for item in 'ABCDE']
>>> # adding 0:A
>>> # adding 1:B
>>> # adding 2:C
>>> # adding 3:D
>>> # adding 4:E
>>>
>>> ## adding arc to the BN
>>> bn.addArc(1,3)
>>> # adding 1->3
>>>
>>> ## removing a node from the BN
>>> bn.erase('C')
>>> # deleting 2
>>> # yes, really deleting 2
```

## ApproximationSchemeListener

## DatabaseGenerationListener

### 1.17.2 Random functions

`pyAgrum.initRandom(seed=0)`

Initialize random generator seed.

#### Parameters

**seed** (*int*) – the seed used to initialize the random generator

#### Return type

None

`pyAgrum.randomProba()`

#### Returns

a random number between 0 and 1 included (i.e. a proba).

**Return type**

float

`pyAgrum.randomDistribution(n)`**Parameters****n** (*int*) – The number of modalities for the ditribution.**Return type**

a random discrete distribution.

### 1.17.3 OMP functions

`pyAgrum.isOMP()`**Returns**

True if OMP has been set at compilation, False otherwise

**Return type**

bool

`pyAgrum.setNumberOfThreads(number)`

To aNone spare cycles (less then 100% CPU occupied), use more threads than logical processors (x2 is a good all-around value).

**Returns****number** – the number of threads to be used**Return type**

int

**Parameters****number** (*int*) –`pyAgrum.getNumberOfLogicalProcessors()`**Return type**

int

`pyAgrum.getNumberOfThreads()`**Return type**

int

## 1.18 Exceptions from aGrUM

**exception** `pyAgrum.GumException(*args)`**args****errorCallStack()****Returns**

the error call stack

**Return type**

str

**errorContent()****Returns**

the error content

**Return type**

str

**errorType()**

**Returns**  
the error type

**Return type**  
str

**what()**

**Return type**  
str

**with\_traceback()**

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

All the exception classes inherit pyAgrum.GumException's functions errorType, errorCallStack and errorContent.

**exception** pyAgrum.DefaultInLabel(\*args)

**args**

**property thisown**  
The membership flag

**exception** pyAgrum.DuplicateElement(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.DuplicateLabel(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.FatalError(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.FormatNotFound(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.GraphError(\*args)

**args**

**property thisown**

The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.IOError(\*args)

**args**

**property thisown**

The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.InvalidArc(\*args)

**args**

**property thisown**

The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.InvalidArgument(\*args)

**args**

**property thisown**

The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.InvalidArgumentsNumber(\*args)

**args**

**property thisown**

The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.InvalidDirectedCycle(\*args)

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.InvalidEdge(\*args)**

args

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.InvalidNode(\*args)**

args

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.NoChild(\*args)**

args

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.NoNeighbour(\*args)**

args

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.NoParent(\*args)**

args

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.NotFound(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.NullElement(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.OperationNotAllowed(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.OutOfBounds(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.ArgumentError(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception pyAgrum.SizeError(\*args)**

**args**

**property thisown**  
The membership flag

**what()**  
Return type  
str

**exception** pyAgrum.SyntaxError(\*args)

**args**

**col()**

**Returns**  
the indice of the colonne of the error

**Return type**  
int

**filename()**

**Return type**  
str

**line()**

**Returns**  
the indice of the line of the error

**Return type**  
int

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.UndefinedElement(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.UndefinedIteratorKey(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.UndefinedIteratorValue(\*args)

**args**

**property thisown**  
The membership flag

**what()**

**Return type**  
str

**exception** pyAgrum.UnknownLabelInDatabase(\*args)

**args**

**property thisown**

The membership flag

**what()**

Return type

str

**exception** pyAgrum.DatabaseError(\*args)

**args**

**property thisown**

The membership flag

**what()**

Return type

str

**exception** pyAgrum.CPTErrror(\*args)

**args**

**property thisown**

The membership flag

**what()**

Return type

str

## 1.19 Configuration for pyAgrum

Configuration for pyAgrum is centralized in an object `gum.config`, singleton of the class `PyAgrumConfiguration`.

**class** pyAgrum.PyAgrumConfiguration(\*args, \*\*kwargs)

`PyAgrumConfiguration` is a the pyAgrum configuration singleton. The configuration is build as a classical `ConfigParser` with read-only structure. Then a value is adressable using a double key: `[section, key]`.

See [this notebook](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/configForPyAgrum.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/configForPyAgrum.ipynb.html>).

### Examples

```
>>> gum.config['dynamicBN', 'default_graph_size']=10
>>> gum.config['dynamicBN', 'default_graph_size']
"10"
```

**add\_hook(fn)**

**diff()**

print the diff between actual configuration and the defaults. This is what is saved in the file `pyagrum.ini` by the method `PyAgrumConfiguration.save()`

**get(section, option)**

Give the value associated to `section.option`. Preferably use `__getitem__` and `__setitem__`.



## Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

### Arguments:

section {str} – the section option {str} – the property

### Returns:

str – the value (as string)

### grep(*search*)

grep in the configuration any section or properties matching the argument. If a section match the argume, all the section is displayed.

### Arguments:

search {str} – the string to find

### load()

load pyagrum.ini in the current directory, and change the properties if needed

### Raises:

FileNotFoundError: if there is no pyagrum.ini in the current directory

### reset()

back to defaults

### run\_hooks()

### save()

Save the diff with the defaults in pyagrum.ini in the current directory

### set(*section, option, value, no\_hook=False*)

set a property in a section. Preferably use `__getitem__` and `__setitem__`.

## Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

### Arguments:



section {str} – the section name (has to exist in defaults) option {str} – the option/property name (has to exist in defaults) value {str} – the value (will be store as string) no\_hook {bool} – (optional) should this call trigger the hooks ?

### Raises:

SyntaxError: if the seccion name or the property name does not exist

## 1.20 Tutorials on pyAgrum

### 1.20.1 Tutorial pyAgrum

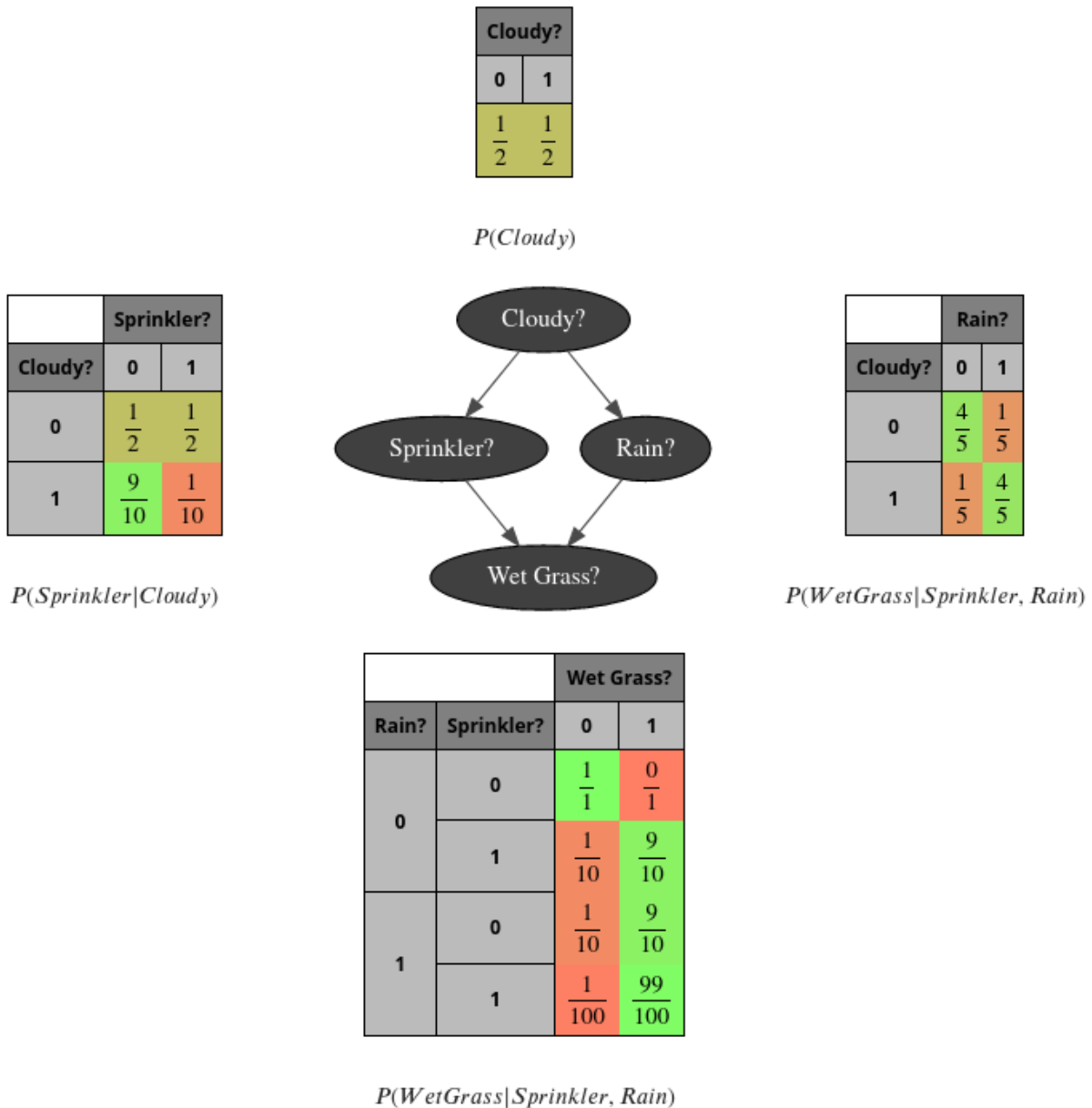
 ( <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a> )	 ( <a href="https://agrum.org">https://agrum.org</a> )	( <a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a> )
--	--	---

```
In [1]: from pylab import *  
import matplotlib.pyplot as plt  
import os
```

### Creating your first Bayesian network with pyAgrum

(This example is based on an OpenBayes [closed] website tutorial)

A Bayesian network (BN) is composed of random variables (nodes) and their conditional dependencies (arcs) which, together, form a directed acyclic graph (DAG). A conditional probability table (CPT) is associated with each node. It contains the conditional probability distribution of the node given its parents in the DAG:



Such a BN allows to manipulate the joint probability  $P(C, S, R, W)$  using this decomposition :

$$P(C, S, R, W) = \prod_X P(X|Parents_X) = P(C) \cdot P(S|C) \cdot P(R|C) \cdot P(W|S, R)$$

Imagine you want to create your first Bayesian network, say for example the ‘Water Sprinkler’ network. This is an easy example. All the nodes are Boolean (only 2 possible values). You can proceed as follows.

## Import the pyAgrum package

```
In [2]: import pyAgrum as gum
```

## Create the network topology

### Create the BN

The next line creates an empty BN network with a 'name' property.

```
In [3]: bn=gum.BayesNet('WaterSprinkler')
print(bn)
BN{nodes: 0, arcs: 0, domainSize: 1, dim: 0}
```

### Create the variables

pyAgrum(aGrUM) provides 4 types of variables :

- LabeledVariable
- RangeVariable
- IntegerVariable
- DiscretizedVariable

In this tutorial, we will use LabeledVariable, which is a variable whose domain is a finite set of labels. The next line will create a variable named 'c', with 2 values and described as 'cloudy?', and it will add it to the BN. The value returned is the id of the node in the graphical structure (the DAG). pyAgrum actually distinguishes the random variable (here the labeledVariable) from its node in the DAG: the latter is identified through a numeric id. Of course, pyAgrum provides functions to get the id of a node given the corresponding variable and conversely.

```
In [4]: c=bn.add(gum.LabeledVariable('c','cloudy ?',2))
print(c)
0
```

You can go on adding nodes in the network this way. Let us use python to compact a little bit the code:

```
In [5]: s, r, w = [ bn.add(name, 2) for name in "srw" ] #bn.add(name, 2) == bn.add(gum.
↳LabeledVariable(name, name, 2))
print (s,r,w)
print (bn)
1 2 3
BN{nodes: 4, arcs: 0, domainSize: 16, dim: 8}
```

## Create the arcs

Now we have to connect nodes, i.e., to add arcs linking the nodes. Remember that `c` and `s` are ids for nodes:

```
In [6]: bn.addArc(c,s)
```

Once again, python can help us :

```
In [7]: for link in [(c,r),(s,w),(r,w)]:
        bn.addArc(*link)
        print(bn)

BN{nodes: 4, arcs: 4, domainSize: 16, dim: 18}
```

pyAgrum provides tools to display `bn` in more user-friendly fashions. Notably, `pyAgrum.lib` is a set of tools written in pyAgrum to help using aGrUM in python. `pyAgrum.lib.notebook` adds dedicated functions for iPython notebook.

```
In [8]: import pyAgrum.lib.notebook as gnb
        bn
```

```
Out[8]: (pyAgrum.BayesNet<double>@0000002A6F47562B0) BN{nodes: 4, arcs: 4, domainSize: 16, dim:
        ↪ 18}
```

## Shorcuts with fastBN

The functions `fast[model]` encode the structure of the graphical model and the type of the variables in a concise language somehow derived from the `dot` language for graphs (see the doc for the underlying method : [fastPrototype](https://pyagrum.readthedocs.io/en/latest/BNModel.html?highlight=fastPrototype#pyAgrum.BayesNet.fastPrototype) (<https://pyagrum.readthedocs.io/en/latest/BNModel.html?highlight=fastPrototype#pyAgrum.BayesNet.fastPrototype>)).

```
In [9]: bn=gum.fastBN("c->r->w<-s<-c")
        bn
```

```
Out[9]: (pyAgrum.BayesNet<double>@0000002A6F4755810) BN{nodes: 4, arcs: 4, domainSize: 16, dim:
        ↪ 18}
```

## Create the probability tables

Once the network topology is constructed, we must initialize the conditional probability tables (CPT) distributions. Each CPT is considered as a Potential object in pyAgrum. There are several ways to fill such an object.

To get the CPT of a variable, use the `cpt` method of your BayesNet instance with the variable's id as parameter.

Now we are ready to fill in the parameters of each node in our network. There are several ways to add these parameters.

### Low-level way

```
In [10]: bn.cpt(c).fillWith([0.4,0.6]) # remember : c= 0
```

```
Out[10]: (pyAgrum.Potential<double>@0000002A6D51B8CF0)
        c
0         | 1         |
-----|-----|
0.4000   | 0.6000   |
```

Most of the methods using a node id will also work with name of the random variable.

```
In [11]: bn.cpt("c").fillWith([0.5,0.5])
Out[11]: (pyAgrum.Potential<double>@000002A6D51B8CF0)
      c      |
0      | 1      |
-----|-----|
0.5000 | 0.5000 |
```

### Using the order of variables

```
In [12]: bn.cpt("s").names
```

```
Out[12]: ('s', 'c')
```

```
In [13]: bn.cpt("s")[:,:] = [ [0.5,0.5], [0.9,0.1] ]
```

Then  $P(S|C = 0) = [0.5, 0.5]$  and  $P(S|C = 1) = [0.9, 0.1]$ .

```
In [14]: print(bn.cpt("s")[1])
[0.9 0.1]
```

The same process can be performed in several steps:

```
In [15]: bn.cpt("s")[0,:]=0.5 # equivalent to [0.5,0.5]
bn.cpt("s")[1,:]=[0.9,0.1]
```

```
In [16]: print(bn.cpt("w").names)
bn.cpt("w")
```

```
('w', 'r', 's')
```

```
Out[16]: (pyAgrum.Potential<double>@000002A6D51B8E30)
      r      | s      | | w      |
0      | 0      | | 0.1595 | 0.8405 |
1      | 0      | | 0.5073 | 0.4927 |
0      | 1      | | 0.4252 | 0.5748 |
1      | 1      | | 0.1210 | 0.8790 |
```

```
In [17]: bn.cpt("w")[0,0,:] = [1, 0] # r=0,s=0
bn.cpt("w")[0,1,:] = [0.1, 0.9] # r=0,s=1
bn.cpt("w")[1,0,:] = [0.1, 0.9] # r=1,s=0
bn.cpt("w")[1,1,:] = [0.01, 0.99] # r=1,s=1
```

### Using a dictionary

This is probably the most convenient way:

```
In [18]: bn.cpt("w")[{ 'r': 0, 's': 0}] = [1, 0]
bn.cpt("w")[{ 'r': 0, 's': 1}] = [0.1, 0.9]
bn.cpt("w")[{ 'r': 1, 's': 0}] = [0.1, 0.9]
bn.cpt("w")[{ 'r': 1, 's': 1}] = [0.01, 0.99]
bn.cpt("w")
```

Out[18]: (pyAgrum.Potential<double>@0000002A6D51B8E30)

	r	s	w	
			0	1
0	0	0	1.0000	0.0000
1	0	0	0.1000	0.9000
0	1	0	0.1000	0.9000
1	1	0	0.0100	0.9900

The use of dictionaries is a feature borrowed from OpenBayes. It facilitates the use and avoid common errors that happen when introducing data into the wrong places.

```
In [19]: bn.cpt("r")[{ 'c':0}]=[0.8,0.2]
bn.cpt("r")[{ 'c':1}]=[0.2,0.8]
```

## Input/output

Now our BN is complete. It can be saved in different format :

```
In [20]: print(gum.availableBNExts())
bif|dsl|net|bifxml|o3prm|uai
```

We can save a BN using BIF format

```
In [21]: gum.saveBN(bn, "out/WaterSprinkler.bif")
```

```
In [22]: with open("out/WaterSprinkler.bif","r") as out:
print(out.read())
```

```
network "unnamedBN" {
// written by aGrUM 1.1.0.9
}

variable c {
    type discrete[2] {0, 1};
}

variable r {
    type discrete[2] {0, 1};
}

variable w {
    type discrete[2] {0, 1};
}

variable s {
    type discrete[2] {0, 1};
}

probability (c) {
    default 0.5 0.5;
}

probability (r | c) {
    (0) 0.8 0.2;
    (1) 0.2 0.8;
}
```

(continues on next page)

(continued from previous page)

```
probability (w | r, s) {
    (0, 0) 1 0;
    (1, 0) 0.1 0.9;
    (0, 1) 0.1 0.9;
    (1, 1) 0.01 0.99;
}
probability (s | c) {
    (0) 0.5 0.5;
    (1) 0.9 0.1;
}
```

```
In [23]: bn2=gum.loadBN("out/WaterSprinkler.bif")
```

We can also save and load it in other formats

```
In [24]: gum.saveBN(bn, "out/WaterSprinkler.net")
with open("out/WaterSprinkler.net", "r") as out:
    print(out.read())
bn3=gum.loadBN("out/WaterSprinkler.net")
```

```
net {
    name = unnamedBN;
    software = "aGrUM 1.1.0.9";
    node_size = (50 50);
}

node c {
    states = (0 1 );
    label = "c";
    ID = "c";
}

node r {
    states = (0 1 );
    label = "r";
    ID = "r";
}

node w {
    states = (0 1 );
    label = "w";
    ID = "w";
}

node s {
    states = (0 1 );
    label = "s";
    ID = "s";
}

potential (c) {
    data = ( 0.5 0.5);
}
```

(continues on next page)



(continued from previous page)

```

potential ( r | c ) {
    data =
    (( 0.8 0.2) % c=0
    ( 0.2 0.8)); % c=1
}

potential ( w | r s ) {
    data =
    ((( 1 0) % s=0 r=0
    ( 0.1 0.9)) % s=1 r=0
    (( 0.1 0.9) % s=0 r=1
    ( 0.01 0.99))); % s=1 r=1
}

potential ( s | c ) {
    data =
    (( 0.5 0.5) % c=0
    ( 0.9 0.1)); % c=1
}

```

## Inference in Bayesian networks

We have to choose an inference engine to perform calculations for us. Many inference engines are currently available in pyAgrum:

- **Exact inference**, particularly :
  - `gum.LazyPropagation` : an exact inference method that transforms the Bayesian network into a hypergraph called a join tree or a junction tree. This tree is constructed in order to optimize inference computations.
  - others: `gum.VariableElimination`, `gum.ShaferShenoy`, ...
- **Samplig Inference** : approximate inference engine using sampling algorithms to generate a sequence of samples from the joint probability distribution (`gum.GibbSSampling`, etc.)
- **Loopy Belief Propagation** : approximate inference engine using inference algorithm exact for trees but not for DAG

```
In [25]: ie=gum.LazyPropagation(bn)
```

## Inference without evidence

```
In [26]: ie.makeInference()
print (ie.posterior("w"))
```

```

w
0      | 1      |
-----|-----|
0.3529 | 0.6471 |

```

```
In [27]: from IPython.core.display import HTML
HTML(f"In our BN, $P(W)={ie.posterior('w')[:]})"
```

```
Out[27]: <IPython.core.display.HTML object>
```

With notebooks, it can be viewed as an HTML table

```
In [28]: ie.posterior("w")[:]
```

```
Out[28]: array([0.3529, 0.6471])
```

## Inference with evidence

Suppose now that you know that the sprinkler is on and that it is not cloudy, and you wonder what is the probability of the grass being wet, i.e., you are interested in distribution  $P(W|S = 1, C = 0)$ . The new knowledge you have (sprinkler is on and it is not cloudy) is called evidence. Evidence is entered using a dictionary. When you know precisely the value taken by a random variable, the evidence is called a hard evidence. This is the case, for instance, when I know for sure that the sprinkler is on. In this case, the knowledge is entered in the dictionary as 'variable name':label

```
In [29]: ie.setEvidence({'s':0, 'c': 0})
ie.makeInference()
ie.posterior("w")
```

```
Out[29]: (pyAgrum.Potential<double>@0000002A6D52EC110)
```

w	
0	1
-----	-----
0.8200	0.1800

When you have incomplete knowledge about the value of a random variable, this is called a soft evidence. In this case, this evidence is entered as the belief you have over the possible values that the random variable can take, in other words, as  $P(\text{evidence}|\text{true value of the variable})$ . Imagine for instance that you think that if the sprinkler is off, you have only 50% chances of knowing it, but if it is on, you are sure to know it. Then, your belief about the state of the sprinkler is [0.5, 1] and you should enter this knowledge as shown below. Of course, hard evidence are special cases of soft evidence in which the beliefs over all the values of the random variable but one are equal to 0.

```
In [30]: ie.setEvidence({'s': [0.5, 1], 'c': [1, 0]})
ie.makeInference()
ie.posterior("w") # using gnb's feature
```

```
Out[30]: (pyAgrum.Potential<double>@0000002A6D52EC370)
```

w	
0	1
-----	-----
0.3280	0.6720

the pyAgrum.lib.notebook utility proposes certain functions to graphically show distributions.

```
In [31]: gnb.showProba(ie.posterior("w"))
nbsphinx-code-borderwhite
```

```
In [32]: gnb.showPosterior(bn, {'s':1, 'c':0}, 'w')
nbsphinx-code-borderwhite
```

### inference in the whole Bayes net

```
In [33]: gnb.showInference(bn, evs={})
nbsphinx-code-borderwhite
```

### inference with hard evidence

```
In [34]: gnb.showInference(bn, evs={'s':1, 'c':0})
nbsphinx-code-borderwhite
```

### inference with soft and hard evidence

```
In [35]: gnb.showInference(bn, evs={'s':1, 'c':[0.3,0.9]})
nbsphinx-code-borderwhite
```

### inference with partial targets

```
In [36]: gnb.showInference(bn, evs={'c':[0.3,0.9]}, targets={'c','w'})
nbsphinx-code-borderwhite
```

## Testing independence in Bayesian networks

One of the strength of the Bayesian networks is to form a model that allows to read qualitative knowledge directly from the graph: the conditional independence. aGrUM/pyAgrum comes with a set of tools to query this qualitative knowledge.

```
In [37]: # fast create a BN (random parameters are chosen for the CPTs)
bn=gum.fastBN("A->B<-C->D->E<-F<-A;C->G<-H<-I->J")
bn
```

```
Out[37]: (pyAgrum.BayesNet<double>@0000002A6F47568000) BN{nodes: 10, arcs: 10, domainSize: 1024,
↳ dim: 46}
```

## Conditional Independence

### Directly

First, one can directly test independence

```
In [38]: def testIndep(bn,x,y,knowning):
    res="" if bn.isIndependent(x,y,knowning) else " NOT"
    giv="." if len(knowning)==0 else f" given {knowning}."
    print(f"{x} and {y} are{res} independent{giv}")

testIndep(bn, "A", "C", [])
testIndep(bn, "A", "C", ["E"])
print()
testIndep(bn, "E", "C", [])
testIndep(bn, "E", "C", ["D"])
print()
testIndep(bn, "A", "I", [])
```

(continues on next page)

(continued from previous page)

```
testIndep(bn,"A","I",["E"])
testIndep(bn,"A","I",["G"])
testIndep(bn,"A","I",["E","G"])
```

A and C are independent.  
A and C are NOT independent given ['E'].

E and C are NOT independent.  
E and C are independent given ['D'].

A and I are independent.  
A and I are independent given ['E'].  
A and I are independent given ['G'].  
A and I are NOT independent given ['E', 'G'].

## Markov Blanket

Second, one can investigate the Markov Blanket of a node. The Markov blanket of a node  $X$  is the set of nodes  $MB(X)$  such that  $X$  is independent from the rest of the nodes given  $MB(X)$ .

```
In [39]: print(gum.MarkovBlanket(bn,"C").toDot())
gum.MarkovBlanket(bn,"C")
```

```
digraph "no_name" {
node [shape = ellipse];
    0[label="A"];
    1[label="B"];
    2[label="C", color=red];
    3[label="D"];
    6[label="G"];
    7[label="H"];

    0 -> 1;
    2 -> 3;
    2 -> 1;
    2 -> 6;
    7 -> 6;

}
```

```
Out[39]: <pyAgrum.pyAgrum.MarkovBlanket; proxy of <Swig Object of type 'gum::MarkovBlanket *'
↪at 0x000002A6F8AF50E0> >
```

```
In [40]: gum.MarkovBlanket(bn,"J")
```

```
Out[40]: <pyAgrum.pyAgrum.MarkovBlanket; proxy of <Swig Object of type 'gum::MarkovBlanket *'
↪at 0x000002A6F8A644B0> >
```

## Minimal conditioning set and evidence Impact using probabilistic inference

For a variable and a list of variables, one can find the sublist that effectively impacts the variable if the list of variables was observed.

```
In [41]: [bn.variable(i).name() for i in bn.minimalCondSet("B",["A","H","J"])]
```

```
Out[41]: ['A']
```

```
In [42]: [bn.variable(i).name() for i in bn.minimalCondSet("B",["A","G","H","J"])]
```

```
Out[42]: ['A', 'G', 'H']
```

This can be also viewed when using `gum.LazyPropagation.evidenceImpact(target,evidence)` which computes  $P(\text{target}|\text{evidence})$  but reduces as much as possible the set of needed evidence for the result :

```
In [43]: ie=gum.LazyPropagation(bn)
ie.evidenceImpact("B",["A","C","H","G"]) # H,G will be removed w.r.t the
↪minimalCondSet above
```

```
Out[43]: (pyAgrum.Potential<double>@000002A6D52EC310)
```

		B	
		0	1
A	C		
0	0	0.6883	0.3117
1	0	0.2372	0.7628
0	1	0.3500	0.6500
1	1	0.5294	0.4706

```
In [44]: ie.evidenceImpact("B",["A","G","H","J"]) # "J" is not necessary to compute the impact
↪of the evidence
```

```
Out[44]: (pyAgrum.Potential<double>@000002A6D52EC1B0)
```

			B	
			0	1
G	H	A		
0	0	0	0.4486	0.5514
1	0	0	0.4965	0.5035
0	1	0	0.5267	0.4733
1	1	0	0.3516	0.6484
0	0	1	0.4442	0.5558
1	0	1	0.4029	0.5971
0	1	1	0.3768	0.6232
1	1	1	0.5280	0.4720

## PS- the complete code to create the first image

```
In [45]: bn=gum.fastBN("Cloudy?->Sprinkler?->Wet Grass?<-Rain?<-Cloudy?")
```

```
bn.cpt("Cloudy?").fillWith([0.5,0.5])
```

```
bn.cpt("Sprinkler?")[:]=[[0.5,0.5],
                        [0.9,0.1]]
```

```
bn.cpt("Rain?") [{'Cloudy?':0}]=[0.8,0.2]
```

```
bn.cpt("Rain?") [{'Cloudy?':1}]=[0.2,0.8]
```

(continues on next page)

(continued from previous page)

```

bn.cpt("Wet Grass?")[{ 'Rain?': 0, 'Sprinkler?': 0}] = [1, 0]
bn.cpt("Wet Grass?")[{ 'Rain?': 0, 'Sprinkler?': 1}] = [0.1, 0.9]
bn.cpt("Wet Grass?")[{ 'Rain?': 1, 'Sprinkler?': 0}] = [0.1, 0.9]
bn.cpt("Wet Grass?")[{ 'Rain?': 1, 'Sprinkler?': 1}] = [0.01, 0.99]

# the next line control the number of visible digits
gum.config['notebook','potential_visible_digits']=2
gnb.sideBySide(bn.cpt("Cloudy?"),captions=['$P(Cloudy)$'])
gnb.sideBySide(bn.cpt("Sprinkler?"),gnb.getBN(bn,size="3!"),bn.cpt("Rain?"),
               captions=['$P(Sprinkler|Cloudy)$',"",'$P(WetGrass|Sprinkler,Rain)$'])
gnb.sideBySide(bn.cpt("Wet Grass?"),captions=['$P(WetGrass|Sprinkler,Rain)$'])

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
<IPython.core.display.HTML object>

```

## PS2- a second glimpse of gum.config

(for more, see the notebook : config for pyAgrum)

```

In [46]: bn=gum.fastBN("Cloudy?->Sprinkler?->Wet Grass?<-Rain?<-Cloudy?")

bn.cpt("Cloudy?").fillWith([0.5,0.5])

bn.cpt("Sprinkler?")[:]=[[0.5,0.5],
                        [0.9,0.1]]

bn.cpt("Rain?")[{ 'Cloudy?':0}]=[0.8,0.2]
bn.cpt("Rain?")[{ 'Cloudy?':1}]=[0.2,0.8]

bn.cpt("Wet Grass?")[{ 'Rain?': 0, 'Sprinkler?': 0}] = [1, 0]
bn.cpt("Wet Grass?")[{ 'Rain?': 0, 'Sprinkler?': 1}] = [0.1, 0.9]
bn.cpt("Wet Grass?")[{ 'Rain?': 1, 'Sprinkler?': 0}] = [0.1, 0.9]
bn.cpt("Wet Grass?")[{ 'Rain?': 1, 'Sprinkler?': 1}] = [0.01, 0.99]



# the next lines control the visualisation of proba as fraction
gum.config['notebook','potential_with_fraction']=True
gum.config['notebook','potential_fraction_with_latex']=True
gum.config['notebook','potential_fraction_limit']=100

gnb.sideBySide(bn.cpt("Cloudy?"),captions=['$P(Cloudy)$'])
gnb.sideBySide(bn.cpt("Sprinkler?"),gnb.getBN(bn,size="3!"),bn.cpt("Rain?"),
               captions=['$P(Sprinkler|Cloudy)$',"",'$P(WetGrass|Sprinkler,Rain)$'])
gnb.sideBySide(bn.cpt("Wet Grass?"),captions=['$P(WetGrass|Sprinkler,Rain)$'])

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
<IPython.core.display.HTML object>

```

## 1.20.2 Using pyAgrum

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org"> (https://agrums.org)</a>	<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html"> (https://agrums.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

```
In [1]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt

import os
```

### Initialisation

- importing pyAgrum
- importing pyAgrum.lib tools
- loading a BN

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
gnb.configuration()

<IPython.core.display.HTML object>
```

```
In [3]: bn=gum.loadBN("res/alarm.dsl")
gnb.showBN(bn,size='9')
nbsphinx-code-borderwhite
```

### Visualisation and inspection

```
In [4]: print(bn.variableFromName('SHUNT'))

SHUNT:Labelized({NORMAL|HIGH})
```

```
In [5]: print(bn.cpt(bn.idFromName('SHUNT')))
```

		SHUNT	
PULMEM INTUBA		NORMAL	HIGH
TRUE	NORMAL	0.1000	0.9000
FALSE	NORMAL	0.9500	0.0500
TRUE	ESOPHA	0.1000	0.9000
FALSE	ESOPHA	0.9500	0.0500
TRUE	ONESID	0.0100	0.9900
FALSE	ONESID	0.0500	0.9500

```
In [6]: gnb.showPotential(bn.cpt(bn.idFromName('SHUNT')),digits=3)
<IPython.core.display.HTML object>
```

## Results of inference

It is easy to look at result of inference

```
In [7]: gnb.showPosterior(bn,{'SHUNT':'HIGH'},'PRESS')
nbsphinx-code-borderwhite
```

```
In [8]: gnb.showPosterior(bn,{'MINVOLSET':'NORMAL'},'VENTALV')
nbsphinx-code-borderwhite
```

Overall results

```
In [9]: gnb.showInference(bn,size="10")
nbsphinx-code-borderwhite
```

What is the impact of observed variables (SHUNT and VENTALV for instance) on another on (PRESS) ?

```
In [10]: ie=gum.LazyPropagation(bn)
ie.evidenceImpact('PRESS',['SHUNT','VENTALV'])
```

```
Out[10]: (pyAgrum.Potential<double>@000001B0648303F0)
```

		PRESS			
VENTAL	SHUNT	ZERO	LOW	NORMAL	HIGH
ZERO	NORMAL	0.0569	0.2669	0.2005	0.4757
LOW	NORMAL	0.0208	0.2515	0.0553	0.6724
NORMAL	NORMAL	0.0769	0.3267	0.1772	0.4192
HIGH	NORMAL	0.0501	0.1633	0.2796	0.5071
ZERO	HIGH	0.0589	0.2726	0.1997	0.4688
LOW	HIGH	0.0318	0.2237	0.0521	0.6924
NORMAL	HIGH	0.1735	0.5839	0.1402	0.1024
HIGH	HIGH	0.0711	0.2347	0.2533	0.4410

## Using inference as a function

It is also easy to use inference as a routine in more complex procedures.

```
In [11]: import time
r=range(0,100,2)
xs=[x/100.0 for x in r]

tf=time.time()
ys=[gum.getPosterior(bn,{'MINVOLSET':[0,x/100.0,0.5]},'VENTALV').tolist()
    for x in r]
delta=time.time()-tf

p=plot(xs,ys)
legend(p,[bn.variableFromName('VENTALV').label(i)
          for i in range(bn.variableFromName('VENTALV').domainSize())],loc=7);
title('VENTALV (100 inferences in %d ms)%delta);
ylabel('posterior Probability');
xlabel('Evidence on MINVOLSET : [0,x,0.5]')
plt.show()
```



nbsphinx-code-borderwhite

Another example : python gives access to a large set of tools. Here the value for the equality of two probabilities of a posterior is easily computed.

```
In [12]: x=[p/100.0 for p in range(0,100)]

tf=time.time()
y=[gum.getPosterior(bn,{ 'HRBP':[1.0-p/100.0,1.0-p/100.0,p/100.0]},'TPR').tolist()
    for p in range(0,100)]
delta=time.time()-tf

p=plot(x,y)
title('HRBP (100 inferences in %d ms)'%delta);
v=bn.variableFromName('TPR');
legend([v.label(i) for i in range(v.domainSize())],loc='best');
np1=(transpose(y)[0]>transpose(y)[2]).argmin()
text(x[np1]-0.05,y[np1][0]+0.005,str(x[np1]),bbox=dict(facecolor='red', alpha=0.1))
plt.show()

nbsphinx-code-borderwhite
```

## BN as a classifier

### Generation of databases

Using the CSV format for the database:

```
In [13]: gum.generateSample(bn,1000,"out/test.csv",with_labels=True)
```

```
Out[13]: -14917.511725169048
```

```
In [14]: with open("out/test.csv","r") as src:
        for _ in range(10):
            print(src.readline(),end="")
```

```
HYPOVOLEMIA, STROKEVOLUME, EXPCO2, CATECHOL, PULMEMBOLUS, VENTTUBE, BP, HRSAT, CO, ANAPHYLAXIS,
→ VENTALV, VENTMACH, HR, PAP, ERRLOWOUTPUT, VENTLUNG, ARTCO2, PCWP, SHUNT, LVEDVOLUME, SAO2, CVP,
→ DISCONNECT, KINKEDTUBE, ERRCAUTER, HISTORY, INSUFFANESTH, MINVOLSET, TPR, HREKG, PRESS,
→ PVSAT, FIO2, INTUBATION, HRBP, MINVOL, LVFAILURE
FALSE, NORMAL, LOW, HIGH, FALSE, LOW, HIGH, HIGH, HIGH, FALSE, NORMAL, NORMAL, HIGH, NORMAL, FALSE,
→ ZERO, NORMAL, NORMAL, HIGH, NORMAL, LOW, NORMAL, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL,
→ NORMAL, HIGH, ZERO, LOW, NORMAL, ONESIDED, HIGH, LOW, FALSE
FALSE, NORMAL, ZERO, HIGH, FALSE, LOW, HIGH, HIGH, HIGH, FALSE, ZERO, NORMAL, HIGH, NORMAL, TRUE,
→ ZERO, LOW, NORMAL, NORMAL, NORMAL, LOW, NORMAL, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL,
→ NORMAL, HIGH, HIGH, LOW, NORMAL, NORMAL, NORMAL, HIGH, FALSE
FALSE, LOW, NORMAL, HIGH, FALSE, NORMAL, LOW, HIGH, LOW, FALSE, HIGH, HIGH, HIGH, NORMAL, FALSE,
→ NORMAL, LOW, NORMAL, NORMAL, NORMAL, NORMAL, NORMAL, TRUE, FALSE, FALSE, FALSE, FALSE, HIGH,
→ HIGH, HIGH, NORMAL, HIGH, NORMAL, NORMAL, HIGH, LOW, FALSE
TRUE, LOW, NORMAL, HIGH, FALSE, HIGH, LOW, HIGH, LOW, FALSE, LOW, HIGH, HIGH, NORMAL, FALSE, LOW,
→ HIGH, HIGH, NORMAL, HIGH, LOW, HIGH, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL, HIGH, HIGH, HIGH,
→ NORMAL, NORMAL, NORMAL, HIGH, NORMAL, FALSE
FALSE, NORMAL, LOW, HIGH, FALSE, ZERO, HIGH, HIGH, HIGH, FALSE, ZERO, NORMAL, HIGH, NORMAL, FALSE,
→ ZERO, HIGH, LOW, NORMAL, LOW, LOW, LOW, TRUE, FALSE, FALSE, FALSE, FALSE, NORMAL, NORMAL, HIGH,
→ LOW, LOW, NORMAL, NORMAL, HIGH, ZERO, FALSE
FALSE, NORMAL, LOW, HIGH, FALSE, LOW, HIGH, HIGH, HIGH, FALSE, ZERO, NORMAL, HIGH, NORMAL, FALSE,
→ ZERO, HIGH, NORMAL, NORMAL, NORMAL, LOW, NORMAL, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL,
→ NORMAL, HIGH, HIGH, LOW, NORMAL, NORMAL, HIGH, ZERO, FALSE
```

(continues on next page)

(continued from previous page)

```

TRUE, LOW, LOW, HIGH, FALSE, LOW, NORMAL, HIGH, NORMAL, FALSE, ZERO, NORMAL, HIGH, NORMAL, FALSE,
→ ZERO, HIGH, HIGH, NORMAL, HIGH, LOW, HIGH, FALSE, FALSE, FALSE, FALSE, TRUE, NORMAL, NORMAL, HIGH,
→ HIGH, LOW, NORMAL, NORMAL, HIGH, ZERO, FALSE
FALSE, NORMAL, LOW, HIGH, FALSE, LOW, LOW, LOW, LOW, FALSE, ZERO, NORMAL, NORMAL, NORMAL, FALSE,
→ ZERO, HIGH, NORMAL, NORMAL, NORMAL, LOW, NORMAL, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL, HIGH,
→ LOW, HIGH, LOW, NORMAL, NORMAL, LOW, ZERO, FALSE
FALSE, NORMAL, LOW, HIGH, FALSE, LOW, HIGH, HIGH, HIGH, FALSE, ZERO, NORMAL, HIGH, NORMAL, FALSE,
→ ZERO, HIGH, NORMAL, NORMAL, NORMAL, LOW, NORMAL, FALSE, FALSE, FALSE, FALSE, FALSE, NORMAL,
→ NORMAL, HIGH, NORMAL, LOW, NORMAL, NORMAL, HIGH, ZERO, FALSE

```

### probabilistic classifier using BN

(because of the use of from-bn-generated csv files, quite good ROC curves are expected)

In [15]: `from pyAgrum.lib.bn2roc import showROC_PR`

```

showROC_PR(bn, "out/test.csv",
            target='CATECHOL', label='HIGH', # class and label
            show_progress=True, show_fig=True, with_labels=True)

```

```

out/test.csv: 100%| |
nbsphinx-code-borderwhite

```

Out[15]: (0.9525796038953934, 0.9300899828, 0.9980437992339163, 0.34813415895)

Using another class variable

In [16]: `showROC_PR(bn, "out/test.csv", 'SA02', 'HIGH', show_progress=True)`

```

out/test.csv: 100%| |
nbsphinx-code-borderwhite

```

Out[16]: (0.9629093016516952, 0.048531175, 0.7758007513206207, 0.5385017134)

### Fast prototyping for BNs

In [17]: `bn1=gum.fastBN("a->b;a->c;b->c;c->d",3)`

```

gnb.sideBySide(*[gnb.getInference(bn1, evs={'c':val}, targets={'a','c','d'}) for val in
→ range(3)],
               captions=[f"Inference given that $c={val}$" for val in range(3)])

```

<IPython.core.display.HTML object>

In [18]: `print(gum.getPosterior(bn1, evs={'c':0}, target='c'))`  
`print(gum.getPosterior(bn1, evs={'c':0}, target='d'))`

*# using pyAgrum.lib.notebook's helpers*

```

gnb.flow.row(gum.getPosterior(bn1, evs={'c':0}, target='c'), gum.getPosterior(bn1, evs={'c'
→ ':0'}, target='d'))

```

```

  c
0  | 1 | 2 |
-----|-----|-----|
1.0000 | 0.0000 | 0.0000 |

```

(continues on next page)

(continued from previous page)

```

      d
0      | 1      | 2      |
-----|-----|-----|
0.6638 | 0.1259 | 0.2103 |

```

&lt;IPython.core.display.HTML object&gt;

**Joint posterior, impact of multiple evidence**

```
In [19]: bn=gum.fastBN("a->b->c->d;b->e->d->f;g->c")
gnb.sideBySide(bn,gnb.getInference(bn))
```

&lt;IPython.core.display.HTML object&gt;

```
In [20]: ie=gum.LazyPropagation(bn)
ie.addJointTarget({"e","f","g"})
ie.makeInference()
gnb.sideBySide(ie.jointPosterior({"e","f","g"}),ie.jointPosterior({"e","g"}),
               captions=["Joint posterior $P(e,f,g)$","Joint posterior $P(e,f)$"])
```

&lt;IPython.core.display.HTML object&gt;

```
In [21]: gnb.sideBySide(ie.evidenceImpact("a",["e","f"]),ie.evidenceImpact("a",["d","e","f"]),
                       captions=["$\\forall e,f, P(a|e,f)$",
                                  "$\\forall d,e,f, P(a|d,e,f)=P(a|d,e)$ using d-separation"]
                       )
```

&lt;IPython.core.display.HTML object&gt;

```
In [22]: gnb.sideBySide(ie.evidenceJointImpact(["a","b"],["e","f"]),ie.evidenceJointImpact(["a
↪","b"],["d","e","f"]),
               captions=["$\\forall e,f, P(a,b|e,f)$",
                          "$\\forall d,e,f, P(a,b|d,e,f)=P(a,b|d,e)$ using d-separation
↪"]
               )
```

&lt;IPython.core.display.HTML object&gt;

In [ ]:

## 1.21 Inference in Bayesian networks

### 1.21.1 Probabilistic Inference with pyAgrum



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

In this notebook, we will show different basic features for probabilistic inference on Bayesian networks using pyAgrum.

First we need some external modules:

```
In [1]: import os

%matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
```

## Basic inference and display

Then we import pyAgrum and the pyAgrum's notebook module, that offers very usefull methods when writting a notebook.

This first example shows how you can load a BayesNet and show it as graph. Note that pyAgrum handles severnal BayesNet file format such as DSL, BIF and UAI.

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
bn=gum.loadBN("res/alarm.dsl")
gnb.showBN(bn,size="9")
nbsphinx-code-borderwhite
```

From there, it is easy to get a posterior using an inference engine :

```
In [3]: ie=gum.LazyPropagation(bn)
ie.makeInference()
print(ie.posterior(bn.idFromName("CATECHOL")))
```

CATECHOL		
NORMAL	HIGH	
-----	-----	
0.0512	0.9488	

But since we are in notebook, why not use pyAgrum notebook's methods ?

```
In [4]: gnb.showPosterior(bn, evs={}, target='CATECHOL')
nbsphinx-code-borderwhite
```

You may also want to see the graph with some posteriors

```
In [5]: # due to matplotlib, format is forced to png.
gnb.showInference(bn, evs={}, targets={"VENTALV", "CATECHOL", "HR", "MINVOLSET"}, size="11")
nbsphinx-code-borderwhite
```

```
In [6]: gnb.showInference(bn,
                        evs={"CO":1, "VENTLUNG":1},
                        targets={"VENTALV",
                                "CATECHOL",
                                "HR",
                                "MINVOLSET",
                                "ANAPHYLAXIS",
                                "STROKEVOLUME",
                                "ERRLOWOUTPUT",
                                "HBR",
```

(continues on next page)

(continued from previous page)

```

        "PULMEMBOLUS",
        "HISTORY",
        "BP",
        "PRESS",
        "CO"},
    size="10")

```

nbsphinx-code-borderwhite

You can even compute all posteriors by leaving the `targets` parameter empty (which is its default value).

```
In [7]: gnb.showInference(bn, evs={"CO":1, "VENTLUNG":1}, size="14")
```

nbsphinx-code-borderwhite

## Showing the information graph

To have a global view of the knowledge brought by the inference, you can also draw the entropy of all nodes

```
In [8]: import pyAgrum.lib.explain as explain
        explain.showInformation(bn, {}, size="14")
```

```
<IPython.core.display.HTML object>
```

... and then observe the impact of an evidence on the whole bayes network :

```
In [9]: explain.showInformation(bn, {"CO":0}, size="9")
```

```
<IPython.core.display.HTML object>
```

## Exploring the junction tree

Lazy Propagation, like several other inference algorithms, uses a junction tree to propagate information.

You can show the junction tree used by Lazy Propagation with pyAgrum:

```
In [10]: jt=ie.junctionTree()
         gnb.showJunctionTree(bn, size="12")
```

nbsphinx-code-borderwhite

```
In [11]: # another representation of the junction, more convenient for investigating the flow
         ↪ of data in the jt
         # the size/width of cliques and separators are proportionnal to the number of nodes
         ↪ in the factor.
         jt.map()
```

```
Out[11]: <pydot.Dot at 0x20f932ea560>
```

## Introspection in junction trees

One can easily walk through the junction tree.

```
In [12]: for n in jt.nodes():
         print([bn.variable(n).name() for n in jt.clique(n)])

['CVP', 'LVEDVOLUME']
['FIO2', 'VENTALV', 'PVSAT']
['ARTCO2', 'EXPCO2', 'VENTLUNG']
```

(continues on next page)

(continued from previous page)

```

['VENTMACH', 'MINVOLSET']
['VENTMACH', 'DISCONNECT', 'VENTTUBE']
['PRESS', 'KINKEDTUBE', 'INTUBATION', 'VENTTUBE']
['ANAPHYLAXIS', 'TPR']
['HRBP', 'ERRLOWOUTPUT', 'HR']
['LVFAILURE', 'HISTORY']
['HREKG', 'HR', 'ERRCAUTER']
['PCWP', 'LVEDVOLUME']
['PAP', 'PULMEMBOLUS']
['SHUNT', 'INTUBATION', 'PULMEMBOLUS']
['HRSAT', 'HR', 'ERRCAUTER']
['LVFAILURE', 'HYPOVOLEMIA', 'LVEDVOLUME']
['HYPOVOLEMIA', 'STROKEVOLUME', 'LVFAILURE']
['CO', 'BP', 'TPR']
['INTUBATION', 'VENTLUNG', 'MINVOL']
['KINKEDTUBE', 'INTUBATION', 'VENTTUBE', 'VENTLUNG']
['INSUFFANESTH', 'TPR', 'ARTCO2', 'SAO2', 'CATECHOL']
['CO', 'STROKEVOLUME', 'HR']
['CO', 'CATECHOL', 'HR']
['CO', 'TPR', 'CATECHOL']
['INTUBATION', 'SHUNT', 'PVSAT', 'SAO2']
['INTUBATION', 'ARTCO2', 'PVSAT', 'SAO2']
['ARTCO2', 'VENTALV', 'INTUBATION', 'PVSAT']
['VENTALV', 'INTUBATION', 'ARTCO2', 'VENTLUNG']

```

```

In [13]: for e in jt.edges():
         print(f"Separator for {e} : {jt.clique(e[0]).intersection(jt.clique(e[1]))}")

```

```

Separator for (13, 30) : {18, 2}
Separator for (2, 33) : {26, 22}
Separator for (3, 4) : {16}
Separator for (26, 27) : {34, 30}
Separator for (7, 26) : {31}
Separator for (12, 13) : {4}
Separator for (31, 32) : {27, 26, 2}
Separator for (23, 31) : {26, 28}
Separator for (5, 22) : {0, 2, 20}
Separator for (17, 24) : {13}
Separator for (19, 27) : {34, 14}
Separator for (24, 26) : {34, 31}
Separator for (32, 33) : {25, 2, 26}
Separator for (6, 23) : {14}
Separator for (23, 27) : {14, 30}
Separator for (11, 16) : {15}
Separator for (10, 14) : {7, 31}
Separator for (8, 17) : {9}
Separator for (0, 16) : {15}
Separator for (1, 32) : {25, 27}
Separator for (20, 33) : {2, 22}
Separator for (4, 22) : {20}
Separator for (14, 26) : {31}
Separator for (22, 33) : {2, 22}
Separator for (30, 31) : {2, 27, 28}
Separator for (16, 17) : {1, 9}

```

```

In [14]: jt.hasRunningIntersection()

```

Out[14]: True

### Introspecting junction trees and friends

The junction tree created by a LazyPropagation is optimized for the query (see RelevanceReasonning notebook). But you can also introspect a junction tree directly from a BN or a graph using the JunctionTreeGenerator's class.

```
In [15]: bn=gum.fastBN("0->1->2<-3->4->5->6<-2->7")
jtg=gum.JunctionTreeGenerator()
gnb.sideBySide(bn,jtg.junctionTree(bn),jtg.eliminationOrder(bn),jtg.
↪binaryJoinTree(bn),
           captions=["A Bayesien network",
                    "a junction tree for this BN",
                    "its elimination order",
                    "an (optimized) binary join tree"])

<IPython.core.display.HTML object>
```

### junction tree from graphs (using uniform domainSize)

```
In [16]: #creating a dag slightly different
dag=bn.dag()
dag.addArc(0,3)
dag.addArc(0,7)
gnb.sideBySide(dag,dag.moralGraph(),jtg.junctionTree(dag),jtg.eliminationOrder(dag),
↪jtg.binaryJoinTree(dag),
           captions=["A DAG","its moral graph",
                    "a junction tree for this dag (with partial order)",
                    "its elimination order (with partial order)",
                    "an (optipmized) binary jointree (with partial order)"])

<IPython.core.display.HTML object>
```

```
In [17]: #creating an undigraph slightly different
ug=bn.dag().moralGraph()
ug.addEdge(0,7)
gnb.sideBySide(ug,jtg.junctionTree(ug),jtg.eliminationOrder(ug),jtg.
↪binaryJoinTree(ug),
           captions=["A undigraph",
                    "a junction tree for this undigraph",
                    "its elimination order",
                    "an (optipmized) binary jointree"])

<IPython.core.display.HTML object>
```

## Using partial order to specify the elimination order

```
In [18]: #adding a partial order for the elimination order
po=[[1,2,3],[0,4,7],[5,6]]
gnb.sideBySide(bn,jtg.junctionTree(bn,po),jtg.eliminationOrder(bn,po),jtg.
↪binaryJoinTree(bn),
          captions=["A Bayesien network",
                    "a junction tree for this BN using partial order",
                    "its elimination order following partial order",
                    "an (optimized) binary join tree"])

<IPython.core.display.HTML object>
```

```
In [19]: #adding a partial order for the elimination order also for the graphs
po=[[0,4,7],[1,3],[5,6,2]]



#creating a dag slightly different
dag=bn.dag()
dag.addArc(0,3)
dag.addArc(0,7)

gnb.sideBySide(dag,dag.moralGraph(),jtg.junctionTree(dag,po),jtg.eliminationOrder(dag,
↪po),jtg.binaryJoinTree(dag,po),
          captions=["A DAG","its moral graph",
                    "a junction tree for this dag (with partial order)",
                    "its elimination order (with partial order)",
                    "an (optipmized) binary jointree (with partial order)"])

<IPython.core.display.HTML object>
```

```
In [ ]:
```

## 1.21.2 Relevance Reasoning with pyAgrum

 ( <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a> )	 ( <a href="https://agrum.org">https://agrum.org</a> )	 ( <a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a> )
--	--	---

Relevance reasoning is the analysis of the influence of evidence on a Bayesian network.

In this notebook we will explain what is relevance reasoning and how to do it using pyAgrum.

```
In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

import time
import os
%matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
```



## Multiple inference

In the well known ‘alarm’ BN, how to analyze the influence on ‘VENTALV’ of a soft evidence on ‘MINVOLSET’ ?

```
In [2]: bn=gum.loadBN("res/alarm.dsl")
gnb.showBN(bn,size="6")
nbsphinx-code-borderwhite
```

We propose to draw the plot of the posterior of ‘VENTALV’ for the evidence :

$$\forall x \in [0, 1], e_{MINVOLSET} = [0, x, 0.5]$$

To do so, we perform a large number of inference and plot the posteriors.

```
In [3]: K=1000
r=range(0,K)
xs=[x/K for x in r]

def getPlot(xs,ys,K,duration):
    p=plot(xs,ys)
    legend(p,[bn.variableFromName('VENTALV').label(i)
              for i in range(bn.variableFromName('VENTALV').domainSize())],loc=7);
    title('VENTALV ({} inferences in {:.5} s)'.format(K,duration));
    ylabel('posterior Probability');
    xlabel('Evidence on MINVOLSET : [0,x,0.5]');
```

## First try : classical lazy propagation

```
In [4]: tf=time.time()
ys=[]
for x in r:
    ie=gum.LazyPropagation(bn)
    ie.setNumberOfThreads(1) # to be fair, we avoid multithreaded inference
    ie.addEvidence('MINVOLSET',[0,x/K,0.5])
    ie.makeInference()
    ys.append(ie.posterior('VENTALV').tolist())
delta1=time.time()-tf
getPlot(xs,ys,K,delta1)
nbsphinx-code-borderwhite
```

The title of the figure above gives the time for those 1000 inference.

## Second try : classical variable elimination

One can note that we just need one posterior. This is a case where VariableElimination should give better results.

```
In [5]: tf=time.time()
ys=[]
for x in r:
    ie=gum.VariableElimination(bn)
    ie.addEvidence('MINVOLSET',[0,x/K,0.5])
    ie.makeInference()
    ys.append(ie.posterior('VENTALV').tolist())
delta2=time.time()-tf
getPlot(xs,ys,K,delta2)
```

nbsphinx-code-borderwhite

pyAgrum give us a function `gum.getPosterior` to do this same job more easily.

```
In [6]: tf=time.time()
ys=[gum.getPosterior(bn,{ 'MINVOLSET':[0,x/K,0.5]}, 'VENTALV').tolist()
      for x in r]
getPlot(xs,ys,K,time.time()-tf)
nbsphinx-code-borderwhite
```

### Last try : optimized Lazy propagation with relevance reasoning and incremental inference

Optimized inference in aGrUM can use the targets and the evidence to optimize the computations. This is called **relevance reasoning**.

Moreover, if the values of the evidence change but not the structure of the query (same nodes as target, same nodes as hard evidence, same nodes as soft evidence), inference in aGrUM may re-use some of the computations from a query to another. This is called **incremental inference**.

```
In [7]: tf=time.time()
ie=gum.LazyPropagation(bn)
ie.setNumberOfThreads(1) # to be fair, we avoid multithreaded inference
ie.addEvidence('MINVOLSET',[1,1,1])
ie.addTarget('VENTALV')
ys=[]
for x in r:
    ie.chgEvidence('MINVOLSET',[0,x/K,0.5])
    ie.makeInference()
    ys.append(ie.posterior('VENTALV').tolist())
delta3=time.time()-tf
getPlot(xs,ys,K,delta3)
nbsphinx-code-borderwhite
```

```
In [8]: print("Mean duration of a lazy propagation          : {:.3f}ms".format(1000*delta1/
↪K))
print("Mean duration of a variable elimination            : {:.3f}ms".format(1000*delta2/
↪K))
print("Mean duration of an optimized lazy propagation : {:.3f}ms".format(1000*delta3/
↪K))

Mean duration of a lazy propagation          : 4.837ms
Mean duration of a variable elimination      : 0.589ms
Mean duration of an optimized lazy propagation : 0.420ms
```

### How it works

```
In [9]: bn=gum.fastBN("Y->X->T1;Z2->X;Z1->X;Z1->T1;Z1->Z3->T2")
ie=gum.LazyPropagation(bn)

gnb.flow.row(bn,bn.cpt("X"),gnb.getJunctionTree(bn),gnb.getJunctionTreeMap(bn,size="3!
↪"),
            captions=["BN","potential","Junction Tree","The map"])

<IPython.core.display.HTML object>
```

aGrUM/pyAgrum uses as much as possible techniques of relevance reasoning to reduce the complexity of the inference.

```
In [10]: ie.setEvidence({"X":0})
gnb.sideBySide(ie,gnb.getDot(ie.joinTree().toDotWithNames(bn)),ie.joinTree().map(),
               captions=["","Join tree optimized for hard evidence on X","the map"])

<IPython.core.display.HTML object>
```

```
In [11]: ie.updateEvidence({"X":[0.1,0.9]})
gnb.sideBySide(ie,gnb.getDot(ie.joinTree().toDotWithNames(bn)),ie.joinTree().map(),
               captions=["","Join tree optimized for soft evidence on X","the map"])

<IPython.core.display.HTML object>
```

```
In [12]: ie.updateEvidence({"Y":0,"X":0,3:[0.1,0.9],"Z1":[0.4,0.6]})
gnb.sideBySide(ie,gnb.getDot(ie.joinTree().toDotWithNames(bn)),ie.joinTree().map(),
               captions=["","Join tree optimized for hard evidence on X and Y, soft
↪on Z2 and Z1","the map"])

<IPython.core.display.HTML object>
```

```
In [13]: ie.setEvidence({"X":0})
ie.setTargets({"T1","Z1"})
gnb.sideBySide(ie,gnb.getDot(ie.joinTree().toDotWithNames(bn)),ie.joinTree().map(),
               captions=["","Join tree optimized for hard evidence on X and targets
↪T1,Z1","the map"])

<IPython.core.display.HTML object>
```

```
In [14]: ie.updateEvidence({"Y":0,"X":0,3:[0.1,0.9],"Z1":[0.4,0.6]})
ie.addJointTarget({"Z2","Z1","T1"})

gnb.sideBySide(ie,
               gnb.getDot(ie.joinTree().toDotWithNames(bn)),ie.joinTree().map(),
               captions=["","Join tree optimized for hard evidence on X and targets
↪T1,Z1","the map"])

<IPython.core.display.HTML object>
```

```
In [15]: ie.makeInference()
ie.jointPosterior({"Z2","Z1","T1"})
```

```
Out[15]: (pyAgrum.Potential<double>@000002BF59732F60)
```

			Z1	
T1	Z2		0	1
-----	-----		-----	-----
0	0		0.0218	0.0111
1	0		0.0125	0.0356
0	1		0.0342	0.2052
1	1		0.0196	0.6599

```
In [16]: ie.jointPosterior({"Z2","Z1"})
```

```
Out[16]: (pyAgrum.Potential<double>@000002BF59732F80)
```

		Z1	
Z2		0	1
-----		-----	-----
0		0.0343	0.0467
1		0.0538	0.8651

```
In [17]: # this will not work
try:
    ie.jointPosterior({"Z3","Z1"})
except gum.UndefinedElement:
    print("Indeed, there is no joint target which contains {4,5} !")
```



Indeed, there is no joint target which contains {4,5} !

```
In [18]: ie.addJointTarget({"Z2","Z1"})
gnb.sideBySide(ie,
               gnb.getDot(ie.joinTree().toDotWithNames(bn)),
               captions=['','JoinTree'])
```

<IPython.core.display.HTML object>

In [ ]:

### 1.21.3 Some other features in Bayesian inference

 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrum.org">https://agrum.org</a>)</p>	<p>(<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>)</p>
--	---	--

Lazy Propagation uses a secondary structure called the “Junction Tree” to perform the inference.

```
In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

bn=gum.loadBN("res/alarm.dsl")
gnb.showJunctionTreeMap(bn);
nbsphinx-code-borderwhite
```

But this junction tree can be transformed to build different probabilistic queries.

```
In [2]: bn=gum.fastBN("A->B->C->D;A->E->D;F->B;C->H")
ie=gum.LazyPropagation(bn)
bn
```

```
Out[2]: (pyAgrum.BayesNet<double>@000000129E82E9A10) BN{nodes: 7, arcs: 7, domainSize: 128,
↪dim: 32}
```

## Evidence impact

Evidence Impact allows the user to analyze the effect of any variables on any other variables

In [3]: `ie.evidenceImpact("B",["A","H"])`

Out[3]: (pyAgrum.Potential<double>@000000129E86B3E30)

		B		
H	A	0	1	
0	0	0.4631	0.5369	
1	0	0.5761	0.4239	
0	1	0.3879	0.6121	
1	1	0.4996	0.5004	

Evidence impact is able to find the minimum set of variables which effectively conditions the analyzed variable

In [4]: `ie.evidenceImpact("E",["A","F","B","D"]) # {A,D,B} d-separates E and F`

Out[4]: (pyAgrum.Potential<double>@000000129E86B3BF0)

			E		
D	B	A	0	1	
0	0	0	0.1907	0.8093	
1	0	0	0.3157	0.6843	
0	1	0	0.1025	0.8975	
1	1	0	0.4230	0.5770	
0	0	1	0.2897	0.7103	
1	0	1	0.4440	0.5560	
0	1	1	0.1651	0.8349	
1	1	1	0.5592	0.4408	

In [5]: `ie.evidenceImpact("E",["A","B","C","D","F"]) # {A,C,D} d-separates E and {B,F}`

Out[5]: (pyAgrum.Potential<double>@000000129E86B3AB0)

			E		
D	A	C	0	1	
0	0	0	0.3251	0.6749	
1	0	0	0.0133	0.9867	
0	1	0	0.4546	0.5454	
1	1	0	0.0229	0.9771	
0	0	1	0.0633	0.9367	
1	0	1	0.4591	0.5409	
0	1	1	0.1047	0.8953	
1	1	1	0.5950	0.4050	

## Evidence Joint Impact

In [6]: `ie.evidenceJointImpact(["A","F"],["B","C","D","E","H"]) # {B,E} d-separates [A,F] and [C,D,H]`

Out[6]: (pyAgrum.Potential<double>@000000129E86B3DB0)

			A		
F	B	E	0	1	
0	0	0	0.0977	0.3931	
1	0	0	0.0170	0.4922	



(continues on next page)

(continued from previous page)

0	1	0	0.0173	0.5420	
1	1	0	0.0696	0.3711	
0	0	1	0.1561	0.3627	
1	0	1	0.0272	0.4541	
0	1	1	0.0282	0.5096	
1	1	1	0.1133	0.3489	

In [ ]:

### 1.21.4 Approximate inference in aGrUM (pyAgrum)

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>	 <a href="https://agrum.org">https://agrum.org</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>
--	--	---

There are several approximate inference for BN in aGrUM (pyAgrum). They share the same API than exact inference.

- Loopy Belief Propagation : LBP is an approximate inference that uses exact calculus methods (when the BN is a tree) even if the BN is not a tree. LBP is a special case of inference : the algorithm may not converge and even if it converges, it may converge to anything (but the exact posterior). LBP however is fast and usually gives not so bad results.
- Sampling inference : Sampling inference use sampling to compute the posterior. The sampling may be (very) slow but those algorithms converge to the exact distribution. aGrUM implements : - Montecarlo sampling, - Weighted sampling, - Importance sampling, - Gibbs sampling.
- Finally, aGrUM propose the so-called 'loopy version' of the sampling algorithms : the idea is to use LBP as a Dirichlet prior for the sampling algorithm. A loopy version of each sampling algorithm is proposed.

```
In [1]: import os

%matplotlib inline
from pylab import *
import matplotlib.pyplot as plt

def unsharpen(bn):
    """
    Force the parameters of the BN not to be a bit more far from 0 or 1
    """
    for nod in bn.nodes():
        bn.cpt(nod).translate(bn.maxParam() / 10).normalizeAsCPT()

def compareInference(ie, ie2, ax=None):
    """
    compare 2 inference by plotting all the points from (posterior(ie), posterior(ie2))
    """
    exact=[]
    appro=[]
    errmax=0
    for node in bn.nodes():
        # potentials as list
        exact+=ie.posterior(node).tolist()
```

(continues on next page)

(continued from previous page)

```

    appro+=ie2.posterior(node).tolist()
    errmax=max(errmax,
               (ie.posterior(node)-ie2.posterior(node)).abs().max())

    if errmax<1e-10: errmax=0
    if ax==None:
        ax=plt.gca() # default axis for plt

    ax.plot(exact,appro,'ro')
    ax.set_title("{} vs {}\n{}\nMax error {:.2.4} in {:.2.4} seconds".format(
        str(type(ie)).split(".")[2].split("_")[0][0:-2], # name of first inference
        str(type(ie2)).split(".")[2].split("_")[0][0:-2], # name of second inference
        ie2.messageApproximationScheme(),
        errmax,
        ie2.currentTime()
    ))

```

```

In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
bn=gum.loadBN("res/alarm.dsl")
unsharpen(bn)

ie=gum.LazyPropagation(bn)
ie.makeInference()

```

```

In [3]: gnb.showBN(bn,size='8')

```

nbsphinx-code-borderwhite

**First, an exact inference.**

```

In [4]: gnb.sideBySide(gnb.getJunctionTreeMap(bn),gnb.getInference(bn,size="8")) # using
↳ LazyPropagation by default
print(ie.posterior("KINKEDTUBE"))

```

&lt;IPython.core.display.HTML object&gt;

KINKEDTUBE		
TRUE	FALSE	
-----	-----	
0.1167	0.8833	

## Gibbs Inference

### Gibbs inference with default parameters

Gibbs inference iterations can be stopped : - by the value of error (epsilon) - by the rate of change of epsilon (MinEpsilonRate) - by the number of iteration (MaxIteration) - by the duration of the algorithm (MaxTime)

```
In [5]: ie2=gum.GibbsSampling(bn)
        ie2.setEpsilon(1e-2)
        gnb.showInference(bn,engine=ie2,size="8")
        print(ie2.posterior("KINKEDTUBE"))
        print(ie2.messageApproximationScheme())
        compareInference(ie,ie2)
nbsphinx-code-borderwhite
```

```
      KINKEDTUBE      |
TRUE      |FALSE      |
-----|-----|
0.1156   | 0.8844   |
```

stopped with rate=0.00673795

nbsphinx-code-borderwhite

With default parameters, this inference has been stopped by a low value of rate.

### Changing parameters

```
In [6]: ie2=gum.GibbsSampling(bn)
        ie2.setMaxIter(1000)
        ie2.setEpsilon(5e-3)
        ie2.makeInference()

        print(ie2.posterior(2))
        print(ie2.messageApproximationScheme())
```

```
      INTUBATION      |
NORMAL    |ESOPHAGEA|ONESIDED |
-----|-----|-----|
0.8264   | 0.1136   | 0.0600   |
```

stopped with max iteration=1000

```
In [7]: compareInference(ie,ie2)
nbsphinx-code-borderwhite
```

```
In [8]: ie2=gum.GibbsSampling(bn)
        ie2.setMaxTime(3)
        ie2.makeInference()

        print(ie2.posterior(2))
        print(ie2.messageApproximationScheme())
        compareInference(ie,ie2)
```

```
      INTUBATION      |
NORMAL    |ESOPHAGEA|ONESIDED |
```

(continues on next page)



(continued from previous page)

```

-----|-----|-----|
0.9700 | 0.0033 | 0.0267 |

stopped with epsilon=0.201897
nbsphinx-code-borderwhite

```

### Looking at the convergence

```

In [9]: ie2=gum.GibbsSampling(bn)
        ie2.setEpsilon(10**-1.8)
        ie2.setBurnIn(300)
        ie2.setPeriodSize(300)
        ie2.setDrawnAtRandom(True)
        gnb.animApproximationScheme(ie2)
        ie2.makeInference()
nbsphinx-code-borderwhite

```

```

In [10]: compareInference(ie,ie2)
nbsphinx-code-borderwhite

```

### Importance Sampling

```

In [11]: ie4=gum.ImportanceSampling(bn)
        ie4.setEpsilon(10**-1.8)
        ie4.setMaxTime(10) #10 seconds for inference
        ie4.setPeriodSize(300)
        ie4.makeInference()
        compareInference(ie,ie4)
nbsphinx-code-borderwhite

```

### Loopy Gibbs Sampling

Every sampling inference has a ‘hybrid’ version which consists in using a first loopy belief inference as a prior for the probability estimations by sampling.

```

In [12]: ie3=gum.LoopyGibbsSampling(bn)

        ie3.setEpsilon(10**-1.8)
        ie3.setMaxTime(10) #10 seconds for inference
        ie3.setPeriodSize(300)
        ie3.makeInference()
        compareInference(ie,ie3)

```

nbsphinx-code-borderwhite

## Comparison of approximate inference

These computations may be a bit long

```
In [13]: def compareAllInference(bn, evs={}, epsilon=10**-1.6, epsilonRate=1e-8, maxTime=20):
    ies=[gum.LazyPropagation(bn),
          gum.LoopyBeliefPropagation(bn),
          gum.GibbsSampling(bn),
          gum.LoopyGibbsSampling(bn),
          gum.WeightedSampling(bn),
          gum.LoopyWeightedSampling(bn),
          gum.ImportanceSampling(bn),
          gum.LoopyImportanceSampling(bn)]

    # burn in for Gibbs samplings
    for i in [2,3]:
        ies[i].setBurnIn(300)
        ies[i].setDrawnAtRandom(True)

    for i in range(2, len(ies)):
        ies[i].setEpsilon(epsilon)
        ies[i].setMinEpsilonRate(epsilonRate)
        ies[i].setPeriodSize(300)
        ies[i].setMaxTime(maxTime)

    for i in range(len(ies)):
        ies[i].setEvidence(evs)
        ies[i].makeInference()

    fig, axes = plt.subplots(1, len(ies)-1, figsize=(35, 3), num='gpplot')
    for i in range(len(ies)-1):
        compareInference(ies[0], ies[i+1], axes[i])
```

## Inference stopped by epsilon

```
In [14]: compareAllInference(bn, epsilon=1e-1)
nbsphinx-code-borderwhite
```

```
In [15]: compareAllInference(bn, epsilon=1e-2)
nbsphinx-code-borderwhite
```

## Inference stopped by time

```
In [16]: compareAllInference(bn, maxTime=1, epsilon=1e-8)
nbsphinx-code-borderwhite
```

```
In [17]: compareAllInference(bn, maxTime=2, epsilon=1e-8)
nbsphinx-code-borderwhite
```

In [ ]:

```
In [4]: def execute(ie):
        with Timer() as t:
            ie.makeInference()
            for i in bn.nodes():
                a=ie.posterior(i)
            return "duration : {:.3f}s".format(t.duration)

def vals(bn,ie):
    exact=[]
    appro=[]
    for node in bn.nodes():
        # potentials as numpy array
        exact+=ie.posterior(node).tolist()

    return exact
```

### Exact inference.

```
In [5]: import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('png')

plt.rcParams["figure.figsize"] = [30,3]

def compareIE(bn,maxtime,epsilon,evs=None):
    ie=gum.LazyPropagation(bn)
    if evs is not None:
        ie.setEvidence(evs)
    x=vals(bn,ie)

    ie2=gum.GibbsSampling(bn)
    if evs is not None:
        ie2.setEvidence(evs)
    ie2.setMaxTime(maxtime)
    ie2.setEpsilon(epsilon)
    txt="Gibbs : "+execute(ie2)+"\n"+ie2.messageApproximationScheme()
    y=vals(bn,ie2)
    plt.subplot(181)
    plt.plot(x,y,'ro')
    plt.title(txt)

    ie3=gum.MonteCarloSampling(bn)
    if evs is not None:
        ie3.setEvidence(evs)
    ie3.setMaxTime(maxtime)
    ie3.setEpsilon(epsilon)
    txt="MonteCarlo : "+execute(ie3)+"\n"+ie3.messageApproximationScheme()
    y=vals(bn,ie3)
    plt.subplot(182)
    plt.plot(x,y,'ro')
    plt.title(txt)

    ie4=gum.WeightedSampling(bn)
    if evs is not None:
        ie4.setEvidence(evs)
    ie4.setMaxTime(maxtime)
```

(continues on next page)

(continued from previous page)

```

ie4.setEpsilon(epsilon)
txt="Weighted : "+execute(ie4)+"\n"+ie4.messageApproximationScheme()
y=vals(bn,ie4)
plt.subplot(183)
plt.plot(x,y,'ro')
plt.title(txt)

ie5=gum.ImportanceSampling(bn)
if evs is not None:
    ie5.setEvidence(evs)
ie5.setMaxTime(maxtime)
ie5.setEpsilon(epsilon)
txt="Importance: "+execute(ie5)+"\n"+ie5.messageApproximationScheme()
y=vals(bn,ie5)
plt.subplot(184)
plt.plot(x,y,'ro')
plt.title(txt)

ie6=gum.LoopyBeliefPropagation(bn)
if evs is not None:
    ie6.setEvidence(evs)
ie6.setMaxTime(maxtime)
ie6.setEpsilon(epsilon)
txt="LBP: "+execute(ie6)+"\n"+ie6.messageApproximationScheme()
y=vals(bn,ie6)
plt.subplot(185)
plt.plot(x,y,'ro')
plt.title(txt)

ie7=gum.LoopyWeightedSampling(bn)
if evs is not None:
    ie7.setEvidence(evs)
ie7.setMaxTime(maxtime)
ie7.setEpsilon(epsilon)
txt="LoopyWeighted: "+execute(ie7)+"\n"+ie7.messageApproximationScheme()
y=vals(bn,ie7)
plt.subplot(186)
plt.plot(x,y,'ro')
plt.title(txt)

ie8=gum.LoopyGibbsSampling(bn)
if evs is not None:
    ie8.setEvidence(evs)
ie8.setMaxTime(maxtime)
ie8.setEpsilon(epsilon)
txt="LoopyGibbs: "+execute(ie8)+"\n"+ie8.messageApproximationScheme()
y=vals(bn,ie8)
plt.subplot(187)
plt.plot(x,y,'ro')
plt.title(txt)

ie9=gum.LoopyImportanceSampling(bn)
if evs is not None:
    ie9.setEvidence(evs)
ie9.setMaxTime(maxtime)
ie9.setEpsilon(epsilon)

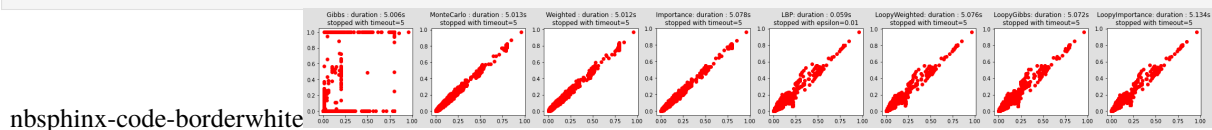
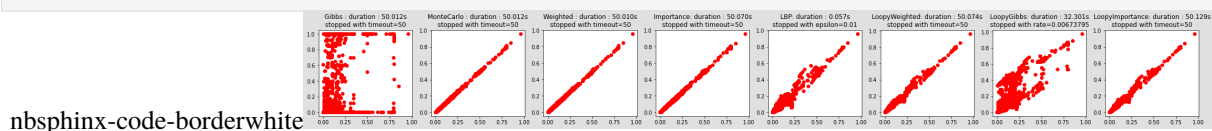
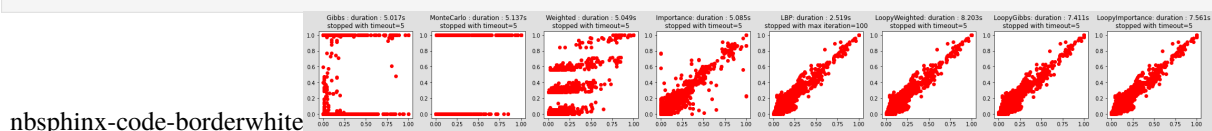
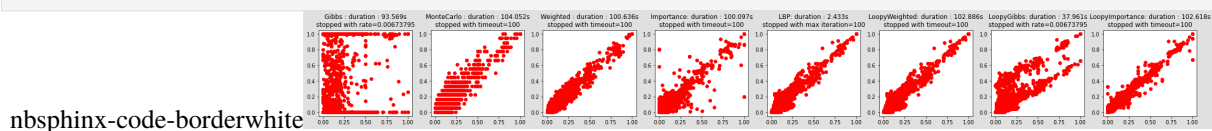
```

(continues on next page)

(continued from previous page)

```
txt="LoopyImportance: "+execute(ie9)+"\n"+ie9.messageApproximationScheme()
y=vals(bn,ie9)
plt.subplot(188)
plt.plot(x,y,'ro')
plt.title(txt)

plt.show()
```

In [6]: `compareIE(bn,5,1e-2)`In [7]: `compareIE(bn,50,1e-2)`In [8]: `compareIE(bn,5,1e-2, evs={'bg_24':0, 'ins_indep_util_23':1, 'renal_cl_14':1})`In [9]: `compareIE(bn,100,1e-2, evs={'bg_24':0, 'ins_indep_util_23':1, 'renal_cl_14':1})`

In [ ]:

## 1.22 Learning Bayesian networks

### 1.22.1 Learning the structure of a Bayesian network



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import os
```

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.explain as explain
import pyAgrum.lib.bn_vs_bn as bnvsbn
```

```
gum.about()
gnb.configuration()
```

```
pyAgrum 1.1.0.9
(c) 2015-2022 Pierre-Henri Wuillemin, Christophe Gonzales
```

```
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, see 'pyAgrum.warranty'.
```

```
<IPython.core.display.HTML object>
```

### Generating the database from a BN

```
In [3]: bn=gum.loadBN("res/asia.bif")
bn
```

```
Out[3]: (pyAgrum.BayesNet<double>@00000022A4F0FBFD0) BN{nodes: 8, arcs: 8, domainSize: 256,
↳ dim: 36}
```

```
In [4]: gum.generateSample(bn,500000,"out/sample_asia.csv",True);
```

```
out/sample_asia.csv: 100%||
```

```
Log2-Likelihood : -1613285.0339585799
```

```
In [5]: with open("out/sample_asia.csv","r") as src:
for _ in range(10):
print(src.readline(),end="")
```

```
tuberculosis,smoking,tuberculos_or_cancer,visit_to_Asia,positive_XraY,bronchitis,lung_
↳ cancer,dyspnoea
1,0,1,1,1,1,1,0
1,0,1,1,1,0,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1
1,0,0,1,0,1,0,0
1,0,1,1,1,1,1,0
1,1,1,1,1,0,1,0
1,1,1,1,1,0,1,0
```

```
In [6]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
print(learner)
```

```
Filename      : out/sample_asia.csv
Size          : (500000,8)
Variables     : visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
↳ XraY[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types : False
Missing values : False
Algorithm     : Greedy Hill Climbing
Score        : BDeu
Prior        : -
```

```
In [7]: print(f"Row of visit_to_Asia : {learner.idFromName('visit_to_Asia')}") # first row is_
↳ 0
```

```
Row of visit_to_Asia : 0
```

```
In [8]: print(f"Variable in row 4 : {learner.nameFromId(4)}")
```

```
Variable in row 4 : lung_cancer
```

The BNlearner is capable of recognizing missing values in databases. For this purpose, just indicate as a last argument the list of the strings that represent missing values.

```
In [9]: # it is possible to add as a last argument a list of the symbols that represent_
↳ missing values:
# whenever a cell of the database is equal to one of these strings, it is considered_
↳ as a
# missing value
learner=gum.BNlearner("res/asia_missing.csv",bn, ['?', 'N/A'] )
print(f"Are there missing values in the database ? {learner.state()['Missing values
↳ '][0]}")
```

```
Are there missing values in the database ? True
```

## type induction

When reading a csv file, BNlearner can try to find the correct type for discrete variable. Especially for numeric values.

```
In [10]: %%writefile out/testTypeInduction.csv
A,B,C,D
1,2,0,hot
0,3,-2,cold
0,1,2,hot
1,2,2,warm
```

```
Overwriting out/testTypeInduction.csv
```

```
In [11]: print("* by default, type induction is on (True) :")
learner=gum.BNlearner("out/testTypeInduction.csv")
bn3=learner.learnBN()
for v in sorted(bn3.names()):
    print(f" - {bn3.variable(v)}")

print("")
print("* but you can disable it :")
learner=gum.BNlearner("out/testTypeInduction.csv",["?"],False)
```

(continues on next page)



(continued from previous page)

```
bn3=learner.learnBN()
for v in sorted(bn3.names()):
    print(f" - {bn3.variable(v)}")

print("")
print("Note that when a Labeled variable is found, the labels are alphabetically_
↳sorted.")
```

```
* by default, type induction is on (True) :
- A:Range([0,1])
- B:Range([1,3])
- C:Integer({-2|0|2})
- D:Labeledized({cold|hot|warm})
```

```
* but you can disable it :
- A:Labeledized({0|1})
- B:Labeledized({1|2|3})
- C:Labeledized({-2|0|2})
- D:Labeledized({cold|hot|warm})
```

Note that when a Labeled variable is found, the labels are alphabetically sorted.

### Parameters learning from the database

We give the *bn* as a parameter for the learner in order to have the variables and the order of the labels for each variables. Please try to remove the argument *bn* in the first line below to see the difference ...

```
In [12]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables_
↳and labels
bn2=learner.learnParameters(bn.dag())
gnb.showBN(bn2)
nbsphinx-code-borderwhite
```

```
In [13]: from IPython.display import HTML

gnb.sideBySide("<H3>Original BN</H3>", "<H3>Learned NB</H3>",
               bn.cpt ('visit_to_Asia'), bn2.cpt ('visit_to_Asia'),
               bn.cpt ('tuberculosis'), bn2.cpt ('tuberculosis'),
               ncols=2)
```

```
<IPython.core.display.HTML object>
```

### Structural learning a BN from the database

Note that, currently, the BNLearner is not yet able to learn in the presence of missing values. This is the reason why, when it discovers that there exist such values, it raises a `gum.MissingValueInDatabase` exception.

```
In [14]: with open("res/asia_missing.csv", "r") as asiafile:
        for _ in range(10):
            print(asiafile.readline(), end="")
        try:
            learner=gum.BNLearner("res/asia_missing.csv", bn, ['?', 'N/A'])
            bn2=learner.learnBN()
        except gum.MissingValueInDatabase:
            print("exception raised: there are missing values in the database")
```

```
smoking, lung_cancer, bronchitis, visit_to_Asia, tuberculosis, tuberculos_or_cancer,
↳ dyspnoea, positive_XraY
0,0,0,1,1,0,0,0
1,1,0,1,1,1,0,1
1,1,1,1,1,1,1,1
1,1,0,1,1,1,0,N/A
0,1,0,1,1,1,1,1
1,1,1,1,1,1,1,1
1,1,1,1,1,1,0,1
1,1,0,1,1,1,0,1
1,1,1,1,1,1,1,1
exception raised: there are missing values in the database
```

## Different learning algorithms

For now, there are three algorithms that are wrapped in pyAgrum : LocalSearchWithTabuList,

```
In [15]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useLocalSearchWithTabuList()
print(learner)
bn2=learner.learnBN()
print("Learned in {0}ms".format(1000*learner.currentTime()))
gnb.flow.row(bn,bn2,explain.getInformation(bn2),captions=["Original BN", "Learned BN",
↳ "information"])
```

Filename	: out/sample_asia.csv
Size	: (500000,8)
Variables	: visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
	↳ XraY[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types	: False
Missing values	: False
Algorithm	: Local Search with Tabu List
Tabu list size	: 2
Score	: BDeu
Prior	: -

Learned in 131.2133ms

<IPython.core.display.HTML object>

To apprehend the distance between the original and the learned BN, we have several tools : - Compute the KL divergence (and other distance) between original and learned joint distribution

```
In [16]: kl=gum.ExactBNdistance(bn,bn2)
kl.compute()
```

```
Out[16]: {'klPQ': 2.111734327848637e-05,
'errorPQ': 0,
'klQP': 1.8016224314088024e-05,
'errorQP': 128,
'hellinger': 0.0027903639478247424,
'bhattacharya': 3.887431170729849e-06,
'jensen-shannon': 5.392839278947e-06}
```

- Compute some scores on the BNs (as binary classifiers) abd show the graphical diff between the two graphs

```
In [17]: gcmp=bnvsbn.GraphicalBNComparator(bn,bn2)
gnb.flow.add(bnvsbn.graphDiff(bn,bn2))
gnb.flow.add(bnvsbn.graphDiffLegend())
gnb.flow.new_line()
gnb.flow.add_html("<br/>".join([f"{k} : {v:.2f}" for k,v in gcmp.skeletonScores().
↪items() if k!='count']),"Skeleton scores")
gnb.flow.add_html("<br/>".join([f"{k} : {v:.2f}" for k,v in gcmp.scores().items() if
↪k!='count']),"Scores")

gnb.flow.display()

<IPython.core.display.HTML object>
```

A greedy Hill Climbing algorithm (with insert, remove and change arc as atomic operations).

```
In [18]: learner=gum.BN Learner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useGreedyHillClimbing()
print(learner)
bn2=learner.learnBN()
print("Learned in {0}ms".format(1000*learner.currentTime()))
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),explain.getInformation(bn2),captions=[
↪"Original BN","Learned BN","Graphical diff","information"])

Filename      : out/sample_asia.csv
Size          : (5000000,8)
Variables     : visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
↪XraY[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types : False
Missing values: False
Algorithm     : Greedy Hill Climbing
Score         : BDeu
Prior        : -

Learned in 108.212ms

<IPython.core.display.HTML object>
```

And a K2 for those who likes it :)

```
In [19]: learner=gum.BN Learner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useK2([0,1,2,3,4,5,6,7])
print(learner)
bn2=learner.learnBN()
print("Learned in {0}ms".format(1000*learner.currentTime()))
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),explain.getInformation(bn2),captions=[
↪"Original BN","Learned BN","Graphical diff","information"])

Filename      : out/sample_asia.csv
Size          : (5000000,8)
Variables     : visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
↪XraY[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types : False
Missing values: False
Algorithm     : K2
K2 order      : visit_to_Asia, tuberculosis, tuberculos_or_cancer, positive_XraY,
↪lung_cancer, smoking, bronchitis, dyspnoea
Score         : BDeu
Prior        : -

Learned in 51.1976ms
```

```
<IPython.core.display.HTML object>
```

K2 can be very good if the order is the good one (a topological order of nodes in the reference)

```
In [20]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useK2([7,6,5,4,3,2,1,0])
print(learner)
bn2=learner.learnBN()
print("Learned in {0}s".format(learner.currentTime()))
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),explain.getInformation(bn2),captions=[
    ↪ "Original BN","Learned BN","Graphical diff","information"])
```

Filename	: out/sample_asia.csv
Size	: (5000000,8)
Variables	: visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
	↪ XraY[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types	: False
Missing values	: False
Algorithm	: K2
K2 order	: dyspnoea, bronchitis, smoking, lung_cancer, positive_XraY,
	↪ tuberculos_or_cancer, tuberculosis, visit_to_Asia
Score	: BDeu
Prior	: -

Learned in 0.065218s

```
<IPython.core.display.HTML object>
```

### Following the learning curve

```
In [21]: import numpy as np
%matplotlib inline

learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useLocalSearchWithTabuList()

# we could prefere a log2likelihood score
# learner.useScoreLog2Likelihood()
learner.setMaxTime(10)

# representation of the error as a pseudo log (negative values really represents,
↪ negative epsilon
@np.vectorize
def pseudolog(x):
    res=np.log(x)#np.log(y)

    return res if x>0 else -res

# in order to control the complexity, we limit the number of parents
learner.setMaxIndegree(7) # no more than 3 parent by node
learner.setEpsilon(1e-10)
gnb.animApproximationScheme(learner,
                             scale=pseudolog) # scale by default is np.log10

bn2=learner.learnBN()
```

nbsphinx-code-borderwhite

## Customizing the learning algorithms

### 1. Learn a tree ?

```
In [22]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useGreedyHillClimbing()

learner.setMaxIndegree(1) # no more than 1 parent by node
print(learner)
bntree=learner.learnBN()
gnb.sideBySide(bn,bntree,gnb.getBNDiff(bn,bntree),explain.getInformation(bntree),
↳captions=["Original BN","Learned BN","Graphical diff","information"])

Filename           : out/sample_asia.csv
Size               : (500000,8)
Variables          : visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2],
↳positive_Xray[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types      : False
Missing values     : False
Algorithm          : Greedy Hill Climbing
Score              : BDeu
Prior              : -
Constraint Max InDegree : 1 (Used only for score-based algorithms.)

<IPython.core.display.HTML object>
```

### 2. with prior structural knowledge

```
In [23]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useGreedyHillClimbing()

# I know that smoking causes cancer
learner.addMandatoryArc("smoking","lung_cancer") # smoking->lung_cancer
# I know that visit to Asia may change the risk of tuberculosis
learner.addMandatoryArc("visit_to_Asia","tuberculosis") # visit_to_Asia->tuberculosis
print(learner)
bn2=learner.learnBN()
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),explain.getInformation(bn2),captions=[
↳"Original BN","Learned BN","Graphical diff","information"])

Filename           : out/sample_asia.csv
Size               : (500000,8)
Variables          : visit_to_Asia[2], tuberculosis[2], tuberculos_or_
↳cancer[2], positive_Xray[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types      : False
Missing values     : False
Algorithm          : Greedy Hill Climbing
Score              : BDeu
Prior              : -
Constraint Mandatory Arcs : {visit_to_Asia->tuberculosis, smoking->lung_cancer}
```

```
<IPython.core.display.HTML object>
```

### 3. changing the scores

By default, a BDEU score is used. But it can be changed.

```
In [24]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
         learner.useGreedyHillClimbing()

         # I know that smoking causes cancer
         learner.addMandatoryArc(0,1)

         # we prefer a log2likelihood score
         learner.useScoreLog2Likelihood()

         # in order to control the complexity, we limit the number of parents
         learner.setMaxIndegree(1) # no more than 1 parent by node
         print(learner)
         bn2=learner.learnBN()
         kl=gum.ExactBNdistance(bn,bn2)
         gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),
                        "<br/>".join(["<b>" + k + "</b> : " + str(v) for k,v in kl.compute().
                        ↪items()])),
                        captions=["original","learned BN","diff","distances"])

Filename           : out/sample_asia.csv
Size               : (500000,8)
Variables          : visit_to_Asia[2], tuberculosis[2], tuberculos_or_
↪cancer[2], positive_Xray[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types      : False
Missing values     : False
Algorithm          : Greedy Hill Climbing
Score              : Log2Likelihood
Prior              : -
Constraint Max InDegree : 1 (Used only for score-based algorithms.)
Constraint Mandatory Arcs : {visit_to_Asia->tuberculosis}

<IPython.core.display.HTML object>
```

### 4. comparing BNs

There are multiple ways to compare Bayes net...

```
In [25]: help(gnb.getBNDiff)

Help on function getBNDiff in module pyAgrum.lib.notebook:

getBNDiff(bn1, bn2, size=None, noStyle=False)
    get a HTML string representation of a graphical diff between the arcs of _bn1_
    ↪(reference) with those of _bn2_.

    if `noStyle` is False use 4 styles (fixed in pyAgrum.config) :
        - the arc is common for both
        - the arc is common but inverted in `bn2`
        - the arc is added in `bn2`
```

(continues on next page)

(continued from previous page)

```

- the arc is removed in `bn2`

Parameters
-----
bn1: pyAgrum.BayesNet
    the reference
bn2: pyAgrum.BayesNet
    the compared one
size: float|str
    size of the rendered graph
noStyle: bool
    with style or not.

Returns
-----
    the HTML representation of the comparison

```

```
In [26]: gnb.showBNDiff(bn,bn2)
         nbsphinx-code-borderwhite
```

```
In [27]: import pyAgrum.lib.bn_vs_bn as gbnbn
         help(gbnbn.graphDiff)
```

```

Help on function graphDiff in module pyAgrum.lib.bn_vs_bn:

graphDiff(bnref, bncmp, noStyle=False)
    Return a pydot graph that compares the arcs of bnref to bncmp.
    graphDiff allows bncmp to have less nodes than bnref. (this is not the case in
    ↪ GraphicalBNComparator.dotDiff())

    if noStyle is False use 4 styles (fixed in pyAgrum.config) :
        - the arc is common for both
        - the arc is common but inverted in _bn2
        - the arc is added in _bn2
        - the arc is removed in _bn2

    See graphDiffLegend() to add a legend to the graph.
    Warning
    -----
    if pydot is not installed, this function just returns None

Returns
-----
pydot.Dot
    the result dot graph or None if pydot can not be imported

```

```
In [28]: gbnbn.GraphicalBNComparator?
```

```

Init signature: gbnbn.GraphicalBNComparator(name1, name2, delta=1e-06)
Docstring:
BNGraphicalComparator allows to compare in multiple way 2 BNs...The smallest
↪ assumption is that the names of the variables are the same in the 2 BNs. But some
↪ comparisons will have also to check the type and domainSize of the variables. The
↪ bns have not exactly the same role : _bn1 is rather the referent model for the
↪ comparison whereas _bn2 is the compared one to the referent model.

```

(continues on next page)

(continued from previous page)

## Parameters

-----

name1 : str or pyAgrum.BayesNet  
 a BN or a filename for reference

name2 : str or pyAgrum.BayesNet  
 another BN or another filename for comparison

**File:** c:\users\phw\scoop\apps\python\current\lib\site-packages\pyagrums\lib\  
 ↪ bn\_vs\_bn.py

**Type:** type

**Subclasses:**

```
In [29]: gcmp=gbnbn.GraphicalBNComparator(bn,bn2)
gnb.sideBySide(bn,bn2,gcmp.dotDiff(),gbnbn.graphDiffLegend(),
               bn2,bn,gbnbn.graphDiff(bn2,bn),gbnbn.graphDiffLegend(),
               ncols=4)
```

&lt;IPython.core.display.HTML object&gt;

```
In [30]: print("But also gives access to different scores :")
print(gcmp.scores())
print(gcmp.skeletonScores())
print(gcmp.hamming())
```

But also gives access to different scores :

```
{'count': {'tp': 4, 'tn': 45, 'fp': 3, 'fn': 4}, 'recall': 0.5, 'precision': 0.
↪ 5714285714285714, 'fscore': 0.5333333333333333, 'dist2opt': 0.6585388898066349}
{'count': {'tp': 6, 'tn': 19, 'fp': 1, 'fn': 2}, 'recall': 0.75, 'precision': 0.
↪ 8571428571428571, 'fscore': 0.7999999999999999, 'dist2opt': 0.2879377767249482}
{'hamming': 3, 'structural hamming': 7}
```

```
In [31]: print("KL divergence can be computed")
kl=gum.ExactBNdistance (bn,bn2)
kl.compute()
```

KL divergence can be computed

```
Out[31]: {'klPQ': 0.12241870078655698,
'errorPQ': 0,
'klQP': 0.03312237743290082,
'errorQP': 64,
'hellinger': 0.20546805471860538,
'bhattacharya': 0.02133452626980637,
'jensen-shannon': 0.024300742692194646}
```

## 5. Mixing algorithms

First we learn a structure with HillClimbing (faster ?)

```
In [32]: learner=gum.BNlearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useGreedyHillClimbing()
learner.addMandatoryArc(0,1)
bn2=learner.learnBN()
kl=gum.ExactBNdistance(bn,bn2)
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),
```

(continues on next page)



(continued from previous page)

```

        "<br/>".join(["<b>" + k + "</b> : " + str(v) for k, v in kl.compute().
→ items()]),
        captions=["original", "learned BN", "diff", "distances"])
<IPython.core.display.HTML object>

```

And then we refine with tabuList

```

In [33]: learner=gum.BNLearner("out/sample_asia.csv",bn) #using bn as template for variables
learner.useLocalSearchWithTabuList()

learner.setInitialDAG(bn2.dag())
print(learner)
bn3=learner.learnBN()
kl=gum.ExactBNdistance(bn,bn3)
gnb.sideBySide(bn,bn2,gnb.getBNDiff(bn,bn2),
        "<br/>".join(["<b>" + k + "</b> : " + str(v) for k, v in kl.compute().
→ items()]),
        captions=["original", "learned BN", "diff", "distances"])

Filename      : out/sample_asia.csv
Size          : (500000,8)
Variables     : visit_to_Asia[2], tuberculosis[2], tuberculos_or_cancer[2], positive_
→ Xray[2], lung_cancer[2], smoking[2], bronchitis[2], dyspnoea[2]
Induced types : False
Missing values : False
Algorithm     : Local Search with Tabu List
Tabu list size : 2
Score        : BDeu
Prior        : -
Initial DAG   : True (digraph {
    0;
    1;
    2;
    3;
    4;
    5;
    6;
    7;

    1 -> 7;
    0 -> 1;
    6 -> 5;
    2 -> 1;
    4 -> 1;
    4 -> 5;
    4 -> 2;
    2 -> 6;
    0 -> 2;
    4 -> 6;
    2 -> 3;
    2 -> 7;
    7 -> 6;
}

)

```

```
<IPython.core.display.HTML object>
```

### Impact of the size of the database for the learning

```
In [34]: import IPython.display
rows=3
sizes=[400,500,700,1000,2000,5000,
        10000,50000,75000,
        100000,150000,175000,
        200000,300000,500000]

def extract_asia(n):
    """
    extract n line from asia.csv to extract.csv
    """
    with open("out/sample_asia.csv","r") as src:
        with open("out/extract_asia.csv","w") as dst:
            for _ in range(n+1):
                print(src.readline(),end="",file=dst)

In [35]: gnb.flow.clear()
nbr=0
l=[]
for i in sizes:
    extract_asia(i)
    learner=gum.BNlearner("out/extract_asia.csv",bn) # using bn as template for
    ↪ variables
    learner.useGreedyHillClimbing()
    print(learner.state()["Size"][0])
    bn2=learner.learnBN()

    kl=gum.ExactBNdistance(bn,bn2)
    r=kl.compute()
    l.append(r['klPQ'])

    gnb.flow.add(gnb.getBNDiff(bn,bn2,size='3!'),f"size={i}")

gnb.flow.display()
plot(sizes,l)
print(l[-1])

(400,8)
(500,8)
(700,8)
(1000,8)
(2000,8)
(5000,8)
(10000,8)
(50000,8)
(75000,8)
(100000,8)
(150000,8)
(175000,8)
(200000,8)
```

(continues on next page)

(continued from previous page)

```
(300000,8)
(500000,8)

<IPython.core.display.HTML object>

1.7210567024329108e-05
nbsphinx-code-borderwhite
```

```
In [36]: gnb.flow.clear()
nbr=0
l=[]
for i in sizes:
    extract_asia(i)
    learner=gum.BN Learner("out/extract_asia.csv",bn) #using bn as template for
    ↪ variables
    learner.useLocalSearchWithTabuList()
    print(learner.state()["Size"][0])
    bn2=learner.learnBN()

    kl=gum.ExactBNdistance(bn,bn2)
    r=kl.compute()
    l.append(r['klPQ'])

    gnb.flow.add(gnb.getBNDiff(bn,bn2,size='3!'),f"size={i}")

gnb.flow.display()
plot(sizes,l)
print(l[-1])
```

```
(400,8)
(500,8)
(700,8)
(1000,8)
(2000,8)
(5000,8)
(10000,8)
(50000,8)
(75000,8)
(100000,8)
(150000,8)
(175000,8)
(200000,8)
(300000,8)
(500000,8)



<IPython.core.display.HTML object>

2.111734327848637e-05
nbsphinx-code-borderwhite
```

In [ ]:

### 1.22.2 Learning BN as probabilistic classifier

Learning a Bayesian network can be used to obtain a classifier for one of the nodes of the model. For more about classifier, see `pyAgrum.skbn`.

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org">https://agrums.org</a>	<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html">https://agrums.gitlab.io/extra/agrum_at_binder.html</a>
--	--	---

```
In [1]: import sys
import os

import numpy as np

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

%matplotlib inline
from pyAgrum.lib.bn2roc import showROC
from pyAgrum.lib.bn2roc import showPR
from pyAgrum.lib.bn2roc import showROC_PR

SIZE_LEARN=10000
SIZE_VALID=2000
```

```
In [2]: bn=gum.loadBN("res/alarm.dsl")
bn
```

```
Out[2]: (pyAgrum.BayesNet<double>@00000026D683B2B20) BN{nodes: 37, arcs: 46, domainSize: 10^16.
↪2389, dim: 752}
```

```
In [3]: gum.generateSample(bn,SIZE_LEARN,"out/learn.csv",show_progress=True,with_labels=True)
gum.generateSample(bn,SIZE_VALID,"out/train.csv",show_progress=True,with_labels=True)
```

```
out/learn.csv: 100%| |
```

```
Log2-Likelihood : -151324.2303803304
```

```
out/train.csv: 100%| |
```

```
Log2-Likelihood : -30435.043192561592
```

```
Out[3]: -30435.043192561592
```

## Learning a BN from learn.csv

```
In [4]: # Learning a BN from the database
learner=gum.BNLearner("out/train.csv")

bn2=learner.useMIIC().learnBN()
currentTime=learner.currentTime()
```

```
In [5]: gnb.flow.add(gnb.getBN(bn2,size="9"),f"Learned with {SIZE_LEARN} lines in
↪{currentTime:.3f}s")
gnb.flow.display()

<IPython.core.display.HTML object>
```

```
In [6]: import pyAgrum.lib.bn_vs_bn as bnvsbn
gnb.flow.add(gnb.getBNDiff(bn,bn2,size="8!"),"Diff with MIIC")
gnb.flow.add(bnvsbn.graphDiffLegend())
gnb.flow.display()

<IPython.core.display.HTML object>
```

```
In [7]: bn3=learner.useGreedyHillClimbing().useNMLCorrection().useScoreBDeu().learnBN()
gnb.flow.add(gnb.getBNDiff(bn,bn3,size="8!"),"Diff with GHC/NMD/BDEU")
gnb.flow.add(bnvsbn.graphDiffLegend())
gnb.flow.display()

<IPython.core.display.HTML object>
```

```
In [8]: bn4=learner.useGreedyHillClimbing().useNMLCorrection().useScoreBDeu().
↪setInitialDAG(bn2.dag()).learnBN()
gnb.flow.add(gnb.getBNDiff(bn,bn4,size="8!"),"Diff with GHC/NMD/BDEU with intial DAG,
↪from MIIC")
gnb.flow.add(bnvsbn.graphDiffLegend())
gnb.flow.display()

<IPython.core.display.HTML object>
```

```
In [9]: print(bn2.names())

{'INSUFFANESTH', 'ERRCAUTER', 'SAO2', 'LVEDVOLUME', 'PAP', 'HREKG', 'BP', 'HISTORY',
↪'HR', 'PCWP', 'CO', 'ERRLOWOUTPUT', 'TPR', 'VENTMACH', 'ARTCO2', 'KINKEDTUBE',
↪'MINVOL', 'HRBP', 'CATECHOL', 'STROKEVOLUME', 'SHUNT', 'VENTLUNG', 'INTUBATION',
↪'ANAPHYLAXIS', 'PVSAT', 'LVFAILURE', 'VENTTUBE', 'MINVOLSET', 'HRSAT', 'PULMEMBOLUS',
↪', 'EXPCO2', 'PRESS', 'VENTALV', 'CVP', 'DISCONNECT', 'HYPOVOLEMIA', 'FIO2'}
```

```
In [10]: gnb.showInference(bn2, evs={}, size="14")
nbsphinx-code-borderwhite
```

## Two classifiers from the learned BN

```
In [11]: print(bn2.variableFromName("HRSAT"))
print(bn2.variableFromName("INTUBATION"))

HRSAT:Labelized({HIGH|LOW|NORMAL})
INTUBATION:Labelized({ESOPHAGEAL|NORMAL|ONESIDED})

In [12]: showROC(bn2,"out/train.csv",'HRSAT','LOW',show_progress=False)
showROC(bn2,"out/train.csv",'HRSAT','NORMAL',show_progress=False)
showROC(bn2,"out/train.csv",'HRSAT','HIGH',show_progress=False);
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite

In [13]: showROC(bn2,"out/train.csv",'INTUBATION',"ESOPHAGEAL",show_progress=False);
nbsphinx-code-borderwhite

In [14]: showPR(bn2,"out/train.csv",'HRSAT','LOW',show_progress=False);
nbsphinx-code-borderwhite

In [15]: showPR(bn2,"out/train.csv",'INTUBATION',"ESOPHAGEAL",show_progress=False);
nbsphinx-code-borderwhite

In [16]: showROC_PR(bn2,"out/train.csv",'HRSAT','LOW',show_progress=False);
nbsphinx-code-borderwhite

In [17]: showROC_PR(bn2,"out/train.csv",'INTUBATION',"ESOPHAGEAL",show_progress=False);
nbsphinx-code-borderwhite

In [ ]:
```

## 1.22.3 Learning essential graphs



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt

import os

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

## Compare learning algorithms

Essentially MIIC and 3off2 computes the essential graph (CPDAG) from data. Essential graphs are mixed graphs.

```
In [2]: learner=gum.BNlearner("out/sample_asia.csv")
learner.use3off2()
learner.useNMLCorrection()
print(learner)
ge3off2=learner.learnEssentialGraph()

Filename      : out/sample_asia.csv
Size          : (5000000,8)
Variables     : tuberculosis[2], smoking[2], tuberculos_or_cancer[2], visit_to_
↳Asia[2], positive_Xray[2], bronchitis[2], lung_cancer[2], dyspnoea[2]
Induced types : True
Missing values : False
Algorithm     : 3off2
Correction    : NML
Prior        : -
```

```
In [3]: gnb.showDot(ge3off2.toDot());
nbsphinx-code-borderwhite
```

```
In [4]: learner=gum.BNlearner("out/sample_asia.csv")
learner.useMIIC()
learner.useNMLCorrection()
print(learner)
gemiic=learner.learnEssentialGraph()
gemiic

Filename      : out/sample_asia.csv
Size          : (5000000,8)
Variables     : tuberculosis[2], smoking[2], tuberculos_or_cancer[2], visit_to_
↳Asia[2], positive_Xray[2], bronchitis[2], lung_cancer[2], dyspnoea[2]
Induced types : True
Missing values : False
Algorithm     : MIIC
Correction    : NML
Prior        : -
```

```
Out[4]: <pyAgrum.pyAgrum.EssentialGraph; proxy of <Swig Object of type 'gum::EssentialGraph *
↳' at 0x0000001E06A662FD0> >
```

For the others methods, it is possible to obtain the essential graph from the learned BN.

```
In [5]: learner=gum.BNlearner("out/sample_asia.csv")
learner.useGreedyHillClimbing()
bnHC=learner.learnBN()
print(learner)
geHC=gum.EssentialGraph(bnHC)
geHC
gnb.sideBySide(bnHC,geHC)

Filename      : out/sample_asia.csv
Size          : (5000000,8)
Variables     : tuberculosis[2], smoking[2], tuberculos_or_cancer[2], visit_to_
↳Asia[2], positive_Xray[2], bronchitis[2], lung_cancer[2], dyspnoea[2]
```

(continues on next page)

(continued from previous page)

```

Induced types : True
Missing values : False
Algorithm      : Greedy Hill Climbing
Score          : BDeu
Prior          : -

```

```
<IPython.core.display.HTML object>
```

```

In [6]: learner=gum.BNlearner("out/sample_asia.csv")
learner.useLocalSearchWithTabuList()
print(learner)
bnTL=learner.learnBN()
geTL=gum.EssentialGraph(bnTL)
geTL
gnb.sideBySide(bnTL,geTL)

```

```

Filename      : out/sample_asia.csv
Size          : (500000,8)
Variables     : tuberculosis[2], smoking[2], tuberculos_or_cancer[2], visit_to_
↳Asia[2], positive_Xray[2], bronchitis[2], lung_cancer[2], dyspnoea[2]
Induced types : True
Missing values : False
Algorithm      : Local Search with Tabu List
Tabu list size : 2
Score          : BDeu
Prior          : -

```

```
<IPython.core.display.HTML object>
```

Hence we can compare the 4 algorithms.

```

In [7]: (
gnb.flow.clear()
.add(ge3off2,"Essential graph from 3off2")
.add(gemiic,"Essential graph from miic")
.add(bnHC,"BayesNet from GHC")
.add(geHC,"Essential graph from GHC")
.add(bnTL,"BayesNet from TabuList")
.add(geTL,"Essential graph from TabuList")
.display()
)



```

```
<IPython.core.display.HTML object>
```

```
In [ ]:
```



### 1.22.4 Dirichlet prior

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org">           (https://agrum.org)         </a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">           (https://agrum.gitlab.io/extra/agrum_at_binder.html)         </a>
--	--	---

#### Dirichlet prior as database

BNLearner gives access of many priors for the parameters and structural learning. One of them is the Dirichlet prior which needs a a prior for every possible parameter in a BN. aGrUM/pyAgrum allows to use a database as a source of Dirichlet prior.

```
In [1]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt

import os

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.explain as explain

sizePrior=30000
sizeData=20000

import os
#the bases will be saved in "out/dirichlet_database.csv" and "out/observaiton_
↪ database.csv"
dirichletDatabase="out/dirichlet_database.csv"
obsDatabase="out/observation_database.csv"
```

#### Generating databases for Dirichlet prior and for the learning

```
In [2]: bnPrior = gum.fastBN("A->B;C;D")
bnData = gum.fastBN("A->B->C->D")
bnData.cpt("B").fillWith([0.99,0.01,
                          0.01,0.99])
bnData.cpt("C").fillWith([0.9,0.1,
                          0.1,0.9])
bnData.cpt("D").fillWith([0.9,0.1,
                          0.1,0.9])
bnPrior.cpt("B").fillWith(bnData.cpt("B"))

gum.generateSample(bnPrior, sizePrior, dirichletDatabase, with_labels=True,random_
↪ order=True)

gum.generateSample(bnData, sizeData, obsDatabase, with_labels=True,random_order=False)
```

(continues on next page)

(continued from previous page)

```
gnb.sideBySide(bnData,bnPrior,
               captions=[f"Database ({sizeData} cases)",f"Prior ({sizePrior} cases)"])
<IPython.core.display.HTML object>
```

## Learning databases

```
In [3]: # bnPrior is used to give the variables and their domains
learnerData = gum.BNLearner(obsDatabase)
learnerPrior = gum.BNLearner(dirichletDatabase)
learnerData.useScoreBIC()
learnerPrior.useScoreBIC()
gnb.sideBySide(learnerData.learnBN(),learnerPrior.learnBN(),
               captions=["Learning from Data","Learning from Prior"])
<IPython.core.display.HTML object>
```

## Learning with Dirichlet prior

Now we use the Dirichlet prior. In order to have an idea of the influence of the priori, we change the weights of Data and Prior from  $[0,1]$  to  $[1,0]$  using a *ratio*  $\in [0,1]$ . The weight of a database is considered equal to the sum of the weights of each row. It is therefore in fact an equivalent sample size that is given.

```
In [4]: learner = gum.BNLearner(obsDatabase, bnPrior)
print(learner)
```

```
Filename      : out/observation_database.csv
Size          : (20000,4)
Variables     : A[2], B[2], C[2], D[2]
Induced types : False
Missing values : False
Algorithm     : Greedy Hill Climbing
Score        : BDeu
Prior        : -
```

```
In [5]: def learnWithRatio(ratio):
        # bnPrior is used to give the variables and their domains

        learner = gum.BNLearner(obsDatabase, bnPrior)
        learner.useDirichletPrior(dirichletDatabase,ratio*sizeData)
        learner.setDatabaseWeight((1-ratio)*sizeData)
        learner.useScoreBIC() # or another score with no included prior
        return learner.learnBN()

ratios=[0.0,0.01,0.05,0.2,0.5,0.8,0.9,0.95,0.99,1.0]
bns=[learnWithRatio(r) for r in ratios]
gnb.sideBySide(*bns,
               captions=[*[f"with ratio {r}<br/> [datasize : {(int(r*sizeData),int((1-
→r)*sizeData))}]" for r in ratios]],
               valign="bottom")
<IPython.core.display.HTML object>
```

The BNs learned when mixing the 2 data sources look much more complex than the data and the Dirichlet structures (with  $ratio \in [0.01, 0.99]$ ). It may seem odd. However, if one looks at the mutual information,

```
In [6]: gnb.sideBySide(*[explain.getInformation(bn) for bn in bns],
                    captions=[f"with ratio {r}<br/> [datasize : {r*sizePrior+(1-
→r)*sizeData}]" for r in ratios]],
                    valign="bottom")

<IPython.core.display.HTML object>
```

It is obvious that these arcs represent weak and spurious correlations due to mixing probabilities (see Wellman et Peacock (99)) that become weaker when the weight of the prior increases.

Another way to look at the mixing is to plot the Kullback-Leibler divergence between the learned BNs and the 2 templates (*bnData* and *bnPrior*)

```
In [7]: def kls(i):
        kl=gum.ExactBNdistance(bnPrior,bns[i])
        y1=kl.compute()
        kl=gum.ExactBNdistance(bnData,bns[i])
        y2=kl.compute()
        return y1['klPQ'],y2['klPQ'],y1['klQP'],y2['klQP']

fig=figure(figsize=(10,6))
ax = fig.add_subplot(1, 1, 1)

x=ratios
y1,y2,y3,y4=zip(*[kls(i) for i in range(len(ratios))])
ax.plot(x,y1,label="M-projection with bnPrior")
ax.plot(x,y3,label="I-projection with bnPrior")
ax.plot(x,y2,label="M-projection with bnData")
ax.plot(x,y4,label="I-projection with bnData")
ax.set_xticks(ratios)
ax.tick_params(rotation=90)
ax.set_xlabel("weight ratio between data and prior")
ax.set_ylabel("KL")
ax.legend(bbox_to_anchor=(0.15, 0.88, 0.7, .102), loc=3,ncol=2, mode="expand",
→borderaxespad=0.)
t=ax.set_title("Weight ratio's Impact on KLS")
plt.show()

nbsphinx-code-borderwhite
```

We can use other divergences (or distances)

```
In [8]: def distances(i):
        kl=gum.ExactBNdistance(bnPrior,bns[i])
        y1=kl.compute()
        kl=gum.ExactBNdistance(bnData,bns[i])
        y2=kl.compute()
        return y1['hellinger'],y2['hellinger'],y1['bhattacharya'],y2['bhattacharya'],y1[
→'jensen-shannon'],y2['jensen-shannon']

fig=figure(figsize=(10,6))
ax = fig.add_subplot(1, 1, 1)

x=ratios
y1,y2,y3,y4,y5,y6=zip(*[distances(i) for i in range(len(ratios))])
```

(continues on next page)

(continued from previous page)

```

ax.plot(x,y1,label="Hellinger with bnPrior")
ax.plot(x,y3,label="Bhattacharya with bnPrior")
ax.plot(x,y5,label="Jensen-Shannon with bnPrior")
ax.plot(x,y2,label="Hellinger with bnData")
ax.plot(x,y4,label="Bhattacharya with bnData")
ax.plot(x,y6,label="Jensen-Shannon with bnData")
ax.set_xticks(ratios)
ax.tick_params(rotation=90)
ax.set_xlabel("weight ratio between data and prior")
ax.set_ylabel("distances")
ax.legend(bbox_to_anchor=(0.15, 0.85, 0.7, .102), loc=3,ncol=2, mode="expand",
↳borderaxespad=0.)
t=ax.set_title("Weight ratio's Impact on distances")
plt.show()
nbsphinx-code-borderwhite

```

Less informative but still possible, we can trace the scores (precision, etc.) from a `pyAgrum.lib.bn_vs_bn.GraphicalBNComparator` (see 07-ComparingBN for more)

```

In [9]: import pyAgrum.lib.bn_vs_bn as gcm

def scores(i):
    cmp=gcm.GraphicalBNComparator(bnPrior,bns[i])
    y1=cmp.scores()
    cmp=gcm.GraphicalBNComparator(bnData,bns[i])
    y2=cmp.scores()
    return y1['recall'],y2['recall'],y1['precision'],y2['precision'],y1['fscore'],
↳y2['fscore'],y1['dist2opt'],y2['dist2opt']

fig=figure(figsize=(20,6))
ax1 = fig.add_subplot(1, 2, 1)
ax2 = fig.add_subplot(1, 2, 2)

x=ratios
y1,y2,y3,y4,y5,y6,y7,y8=zip(*[scores(i) for i in range(len(ratios))])
ax1.plot(x,y1,label="recall with bnPrior")
ax1.plot(x,y3,label="precision with bnPrior")
ax1.plot(x,y5,label="fscore with bnPrior")
ax1.plot(x,y7,label="dist2opt with bnPrior")

ax2.plot(x,y2,label="recall with bnData")
ax2.plot(x,y4,label="precision with bnData")
ax2.plot(x,y6,label="fscore with bnData")
ax2.plot(x,y8,label="dist2opt with bnData")

ax1.set_xticks(ratios)
ax1.tick_params(rotation=90)
ax1.set_xlabel("weight ratio between data and prior")
ax1.set_ylabel("KL")
ax1.legend(bbox_to_anchor=(0.15, 0.88, 0.7, .102), loc=3,ncol=2, mode="expand",
↳borderaxespad=0.)
ax1.set_title("Weight ratio's Impact on scores")

ax2.set_xticks(ratios)
ax2.tick_params(rotation=90)
ax2.set_xlabel("weight ratio between data and prior")

```

(continues on next page)

(continued from previous page)

```
ax2.set_ylabel("KL")
ax2.legend(bbox_to_anchor=(0.15, 0.88, 0.7, .102), loc=3, ncol=2, mode="expand",
           borderaxespad=0.)
ax2.set_title("Weight ratio's Impact on scores")

plt.show()
nbsphinx-code-borderwhite
```

## Weighted database and records

Database can be weighted as done above. But you can also fix the weight record by record in the database. Note that the weight of database is the sum of all weights for each record. And then

```
learner.setDatabaseWeight(2.5)
```

is equivalent to

```
siz=learner.nbRows()
for i in range(siz):
    learner.setRecordWeight(i,2.5/siz)
```

```
In [10]: bn=gum.fastBN("X->Y")

#the base will be saved in basefile="out/dataW.csv"
basefile="out/dataW.csv"
```

In the 2 next cells, we compute the parameters of bn using 2 bases : in the next cell, the base contains 8 rows of weight 1. In the next one, the base contains only 4 rows but two of them have different weights. The sum of the weights in this second base is 8 as well ...

So the parameters are exactly the same.

```
In [11]: %%writefile 'out/dataW.csv'
X,Y
1,0
0,1
0,1
0,0
1,0
0,1
1,1
0,1
```

Overwriting out/dataW.csv

```
In [12]: learner=gum.BNLearner(basefile)
bn1=learner.learnParameters(bn.dag())
gnb.flow.row(bn1.cpt("X"),bn1.cpt("Y"))

<IPython.core.display.HTML object>
```

```
In [13]: %%writefile 'out/dataW.csv'
X,Y
0,0
1,0
0,1
1,1
```

```
Overwriting out/dataW.csv
```



```
In [14]: learner.setRecordWeight(1,2.0)
learner.setRecordWeight(2,4.0)

bn2=learner.learnParameters(bn.dag())
gnb.flow.row(bn2.cpt("X"),bn2.cpt("Y"))

<IPython.core.display.HTML object>
```

```
In [ ]:
```

### 1.22.5 Parametric EM (missing data)

 ( <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a> )	 ( <a href="https://agrum.org">https://agrum.org</a> )	( <a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a> )
--	--	---

```
In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

import os
#the bases will be saved in "out/*.csv"
EMnomissing="out/EM_nomissing.csv"
EMmissing="out/EM_missing.csv"
```

#### Generating data with missing values (at random)

```
In [2]: src=gum.fastBN("A->B<-C->D->E<-B;D->F")
gum.generateSample(src,5000,EMnomissing,random_order=False)
src
```

```
Out[2]: (pyAgrum.BayesNet<double>@00000021FF6B63560) BN{nodes: 6, arcs: 6, domainSize: 64, dim:
↪ 28}
```

```
In [3]: import pandas as pd
import numpy as np

def add_missing(src,dst,proba):
    df=pd.read_csv(src)
    mask=np.random.choice([True, False], size=df.shape,p=[proba,1-proba])
    df.mask(mask).to_csv(dst,na_rep='?',index=False,float_format='%.0f')

gum.generateSample(src,5000,EMnomissing,random_order=False)
add_missing(EMnomissing,EMmissing,proba=0.1)
```

```
In [4]: print("No missing")
with open(EMnomissing,"r") as srcfile:
    for _ in range(10):
        print(srcfile.readline(),end="")
print("Missing")
with open(EMmissing,"r") as srcfile:
    for _ in range(10):
        print(srcfile.readline(),end="")
```

```
No missing
A,B,C,D,E,F
1,1,1,0,0,1
1,1,0,1,1,1
0,1,1,0,0,0
1,1,0,0,1,1
1,1,1,0,1,0
0,1,1,0,1,1
1,1,1,0,0,1
1,0,0,1,0,0
1,0,1,0,1,1
Missing
A,B,C,D,E,F
1,1,?,0,?,1
1,1,0,1,1,1
?,?,1,0,?,0
?,1,?,0,1,1
1,1,1,0,1,0
0,1,1,0,1,1
1,1,1,0,0,1
1,0,0,1,0,?
1,0,?,0,1,1
```

### Learning with missing data

```
In [5]: learner = gum.BNLearner(EMmissing,src, ["?"])
print(f"Missing values in {EMmissing} : {learner.hasMissingValues()}")
Missing values in out/EM_missing.csv : True
```

```
In [6]: try:
    learner.learnParameters(src.dag())
except gum.MissingValueInDatabase:
    print("Learning is not possible without EM if there are some missing values.")
Learning is not possible without EM if there are some missing values.
```

```
In [7]: learner.useEM(1e-3)
learner.useSmoothingPrior()
print(learner)
bn=learner.learnParameters(src.dag())
print(f"# iterations : {learner.nbrIterations()}")
gnb.flow.row(gnb.getInference(src),gnb.getInference(bn))

Filename      : out/EM_missing.csv
Size          : (5000,6)
Variables     : A[2], B[2], C[2], D[2], E[2], F[2]
Induced types : False
```

(continues on next page)

(continued from previous page)

```
Missing values : True
Algorithm      : Greedy Hill Climbing
Score         : BDeu
Prior         : Smoothing (The BDeu score already contains a different 'implicit' prior. Therefore, the learning will probably be biased.)
Prior weight   : 1.000000
EM            : True
EM epsilon    : 0.001000

# iterations : 6

<IPython.core.display.HTML object>
```

### Learning with smaller error (and no smoothing)

```
In [8]: learner = gum.BNLearner(EMmissing,src, ["?"])
learner.setVerbosity(True)
learner.useEM(1e-8)
bn2=learner.learnParameters(src.dag())
gnb.flow.row(gnb.getInference(src),gnb.getInference(bn2),captions=["Source",f
↪ "Estimation EM en {learner.nbrIterations()} iteration(s)"])

<IPython.core.display.HTML object>
```

```
In [9]: import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(1,1+learner.nbrIterations()),learner.history())
plt.xticks(np.arange(1, 1+learner.nbrIterations(), step=2))
plt.title("Error during EM iterations");
nbsphinx-code-borderwhite
```

In [ ]:

### 1.22.6 Scores, Chi2, etc. with BNLearner



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: import os

import pyAgrum as gum
import pyAgrum.lib.notebook as gn
```



## Generating the database for scoring

```
In [2]: bn=gum.loadBN("res/asia.bif")
bn

Out[2]: (pyAgrum.BayesNet<double>@0000001C4B7746690) BN{nodes: 8, arcs: 8, domainSize: 256,
↳ dim: 36}

In [3]: # we create a quite large database
gum.generateSample(bn,500000,"out/sample_score.csv",False)

Out[3]: -1615826.0306111004
```

## Testing d-separations using chi2 in the database

```
In [4]: # do not forget that the generation process above is random : from time to time, the
↳ tests my not be correct...
def isIndep(pvalue):
    return pvalue>=0.05

def testIndepFromChi2(learner,var1,var2,kno=[]):
    """
    Just prints the resultat of the chi2
    """
    stat,pvalue=learner.chi2(var1,var2,kno)
    if len(kno)==0:
        print("From Chi2 tests, is '{}' indep from '{}' ==> {}".format(var1,var2,
↳ isIndep(pvalue)))
    else:
        print("From Chi2 tests, is '{}' indep from '{}' given {} : {}".format(var1,
↳ var2,kno,isIndep(pvalue)))

learner=gum.BNLearner("out/sample_score.csv")

testIndepFromChi2(learner,"visit_to_Asia","smoking")
testIndepFromChi2(learner,"visit_to_Asia","smoking",['tuberculos_or_cancer'])
testIndepFromChi2(learner,"visit_to_Asia","smoking",['positive_XraY'])
testIndepFromChi2(learner,"dyspnoea","smoking")
testIndepFromChi2(learner,"dyspnoea","smoking",["lung_cancer","bronchitis"])

From Chi2 tests, is 'visit_to_Asia' indep from 'smoking' ==> True
From Chi2 tests, is 'visit_to_Asia' indep from 'smoking' given ['tuberculos_or_cancer
↳ '] : False
From Chi2 tests, is 'visit_to_Asia' indep from 'smoking' given ['positive_XraY'] :
↳ False
From Chi2 tests, is 'dyspnoea' indep from 'smoking' ==> False
From Chi2 tests, is 'dyspnoea' indep from 'smoking' given ['lung_cancer', 'bronchitis
↳ '] : True
```

## Evolution of chi2 p-values w.r.t the size of the database (in Asia)

```
In [5]: def consolidationIndepFromChi2(bn,size,lindep,nbr=20):
        """
        Using $nbr$ generated databases of size $size$ from the bn $bn$,
        consolidate the p-value for a list $lindep$ of conditional independence to test.

        return the list of consolidated pValues
        """
        pvalue_cumul=[0.0]*len(lindep)
        for i in range(nbr):
            gum.generateSample(bn,size,"out/sample_score.csv",False)
            learner=gum.BNlearner("out/sample_score.csv")
            for i,(var1,var2,kno) in enumerate(lindep):
                stat,pvalue=learner.chi2(var1,var2,kno)
                pvalue_cumul[i]+=pvalue
        return [p/nbr for p in pvalue_cumul]

sizes=[50,100,500,1000,2000,5000,10000,20000,50000,100000,200000]
pvalues1,pvalues2,pvalues3,pvalues4,pvalues5,
↪ pvalues6=zip(*[consolidationIndepFromChi2(bn,siz,
        [("visit_to_Asia","smoking",['tuberculos_or_
↪ cancer']),
        ("visit_to_Asia","smoking",[]),
        ("dyspnoea","smoking",[]),
        ("dyspnoea","smoking",["lung_cancer","bronchitis
↪ ]),
        ("tuberculosis","bronchitis",[]),
        ("tuberculosis","bronchitis",["dyspnoea"])]])
        for siz in sizes])
```

```
In [6]: %matplotlib inline
        from pylab import *
        import matplotlib.pyplot as plt
        import matplotlib.patches as patches

        fig=figure(figsize=(10,6))
        ax = fig.add_subplot(1, 1, 1)

        ax.plot(sizes,pvalues1,label="NOT(A indep S given TorC)", linestyle='dashed')
        ax.plot(sizes,pvalues2,label="A indep S")
        ax.plot(sizes,pvalues3,label="NOT(D indep S)", linestyle='dashed')
        ax.plot(sizes,pvalues4,label="D indep S given L,B")
        ax.plot(sizes,pvalues5,label="T indep B")
        ax.plot(sizes,pvalues6,label="NOT(T indep B given D)", linestyle='dashed')

        ax.tick_params(rotation=90)
        ax.set_xlabel("data size")
        ax.set_ylabel("pValue")
        ax.legend(bbox_to_anchor=(0.15, 0.88, 0.7, .102), loc=3,ncol=3, mode="expand",
↪ borderaxespad=0.)

        rect = patches.Rectangle((0,0),max(sizes),0.05,linewidth=1,edgecolor='#FF8888',
↪ facecolor='#FF8888')
        ax.add_patch(rect)
        ax.annotate("Critical region",xytext=(190000,0.2),xy=(190000,0.05),
```

(continues on next page)

(continued from previous page)

```

        ha="right", va="center",
        arrowprops=dict(arrowstyle="->",
                        connectionstyle="arc3,rad=-0.15"
                        ),
        bbox=dict(boxstyle="square", fc="w"))

ax.set_title("Chi2 pvalue=f(datasize)")
gnb.flow.add(fig)
gnb.flow.add(gnb.getBN(gum.fastBN("A->T->TorC->X;S->C->TorC->D<-B<-S")))
gnb.flow.new_line()

fig

```

nbsphinx-code-border style="border: 1px solid #ccc; padding: 10px; margin: 10px 0;">

### Testing d-separations using G2 in the database

```

In [7]: def testIndepFromG2(learner, var1, var2, kno=[]):
        """
        Just prints the resultat of the G2
        """
        stat, pvalue = learner.G2(var1, var2, kno)
        if len(kno) == 0:
            print("From G2 tests, is '{}' indep from '{}' ==> {}".format(var1, var2,
↪ isIndep(pvalue)))
        else:
            print("From G2 tests, is '{}' indep from '{}' given {} : {}".format(var1, var2,
↪ kno, isIndep(pvalue)))

learner = gum.BNLearner("out/sample_score.csv")

testIndepFromG2(learner, "visit_to_Asia", "smoking")
testIndepFromG2(learner, "visit_to_Asia", "smoking", ['tuberculos_or_cancer'])
testIndepFromG2(learner, "visit_to_Asia", "smoking", ['positive_XraY'])
testIndepFromG2(learner, "dyspnoea", "smoking")
testIndepFromG2(learner, "dyspnoea", "smoking", ["lung_cancer", "bronchitis"])

From G2 tests, is 'visit_to_Asia' indep from 'smoking' ==> True
From G2 tests, is 'visit_to_Asia' indep from 'smoking' given ['tuberculos_or_cancer'] ↪
↪ : False
From G2 tests, is 'visit_to_Asia' indep from 'smoking' given ['positive_XraY'] : True
From G2 tests, is 'dyspnoea' indep from 'smoking' ==> False
From G2 tests, is 'dyspnoea' indep from 'smoking' given ['lung_cancer', 'bronchitis'] ↪
↪ : True

```

## Evolution of G2 p-values w.r.t the size of the database (in Asia)

```
In [8]: def consolidationIndepFromG2(bn,size,lindep,nbr=20):
        """
        Using $nbr$ generated databases of size $size$ from the bn $bn$,
        consolidate the p-value for a list $lindep$ of conditional independence to test.

        return the list of consolidated pValues
        """
        pvalue_cumul=[0.0]*len(lindep)
        for i in range(nbr):
            gum.generateSample(bn,size,"out/sample_chi2.csv",False)
            learner=gum.BNlearner("out/sample_chi2.csv")
            for i,(var1,var2,kno) in enumerate(lindep):
                stat,pvalue=learner.G2(var1,var2,kno)
                pvalue_cumul[i]+=pvalue
        return [p/nbr for p in pvalue_cumul]

        sizes=[50,100,500,1000,2000,5000,10000,20000,50000,100000,200000]
        pvalues1,pvalues2,pvalues3,pvalues4,pvalues5,
        ↪ pvalues6=zip(*[consolidationIndepFromG2(bn,siz,
        ↪ [("visit_to_Asia","smoking",['tuberculos_or_cancer']),
        ↪
        ↪ ("visit_to_Asia","smoking",[]),
        ↪
        ↪ ("dyspnoea","smoking",[]),
        ↪
        ↪ ("dyspnoea","smoking",["lung_cancer","bronchitis"]),
        ↪
        ↪ ("tuberculosis","bronchitis",[]),
        ↪
        ↪ ("tuberculosis","bronchitis",["dyspnoea"])]))
        for siz in sizes])
```

```
In [9]: fig=figure(figsize=(10,6))
        ax = fig.add_subplot(1, 1, 1)

        ax.plot(sizes,pvalues1,label="NOT(A indep S given TorC)", linestyle='dashed')
        ax.plot(sizes,pvalues2,label="A indep S")
        ax.plot(sizes,pvalues3,label="NOT(D indep S)", linestyle='dashed')
        ax.plot(sizes,pvalues4,label="D indep S given L,B")
        ax.plot(sizes,pvalues5,label="T indep B")
        ax.plot(sizes,pvalues6,label="NOT(T indep B given D)", linestyle='dashed')

        ax.tick_params(rotation=90)
        ax.set_xlabel("data size")
        ax.set_ylabel("pValue")
        ax.legend(bbox_to_anchor=(0.15, 0.83, 0.7, .102), loc=3,ncol=2, mode="expand",
        ↪ borderaxespad=0.)

        rect = patches.Rectangle((0,0),max(sizes),0.05,linewidth=1,edgecolor='#FF8888',
        ↪ facecolor='#FF8888')
        ax.add_patch(rect)
        ax.annotate("Critical region",xytext=(190000,0.2),xy=(190000,0.05),
        ↪ ha="right", va="center",
        ↪ arrowprops=dict(arrowstyle="->",
```

(continues on next page)

(continued from previous page)

```

        connectionstyle="arc3,rad=-0.15"
    ),
    bbox=dict(boxstyle="square", fc="w"))

ax.set_title("G2 pvalue=f(datasize)")
gnb.flow.add(fig)
gnb.flow.add(gnb.getBN(gum.fastBN("A->T->TorC->X;S->C->TorC->D<-B<-S")))
gnb.flow.new_line()

```

nbsphinx-code-border style

## Conditional joint log-likelihood

With BNLearner, you can also check the joint (conditional) log-likelihood in the base

In [10]: bn

Out[10]: (pyAgrum.BayesNet<double>@0000001C4B7746690) BN{nodes: 8, arcs: 8, domainSize: 256,   
 ↳ dim: 36}

```

In [11]: siz=10000
gum.generateSample(bn,siz,"out/sample_score.csv",False)
learner=gum.BNLearner("out/sample_score.csv")

def affLL(learner,s1,s2=[]):
    if len(s2)==0:
        print("{} : {}".format(s1,learner.logLikelihood(s1)))
    else:
        print("{}|{} : {}".format(s1,s2,learner.logLikelihood(s1,s2)))

def dsepByLL(learner,x,y,z): # is X indep of Y given Z ?
    lxy_z=learner.logLikelihood([x,y],[z])
    lx_z=learner.logLikelihood([x],[z])
    ly_z=learner.logLikelihood([y],[z])
    print("{} indep {} given {} : {}".format(x,y,z,lxy_z-lx_z-ly_z))

print("Conditional Joint LogLikelihood")
affLL(learner,["lung_cancer","bronchitis","smoking"])
affLL(learner,["smoking"])
affLL(learner,["lung_cancer","bronchitis"],["smoking"])

print("-----")
print("LL indep test")
dsepByLL(learner,"lung_cancer","bronchitis","smoking")
dsepByLL(learner,"tuberculos_or_cancer","bronchitis","dyspnoea")

Conditional Joint LogLikelihood
['lung_cancer', 'bronchitis', 'smoking'] : -22142.063042301077
['smoking'] : -9995.844639313984
['lung_cancer', 'bronchitis']|['smoking'] : -12146.218402987093
-----
LL indep test
lung_cancer indep bronchitis given smoking : 0.4046632959161798
tuberculos_or_cancer indep bronchitis given dyspnoea : 160.49141588867315

```

## Evolution of conditional log-likelihood w.r.t the size of the database (in Asia)

```
In [12]: def consolidationIndepFromLL(bn,size,lindep,nbr=20):
        """
        Using $nbr$ generated databases of size $size$ from the bn $bn$,
        consolidate the logLikelihoos for a list $lindep$ of conditional independence to_
        ↪ test.

        return the list of consolidated pValues
        """
        LL_cumul=[0.0]*len(lindep)
        for i in range(nbr):
            gum.generateSample(bn,size,"out/sample_score.csv",False)
            learner=gum.BNlearner("out/sample_score.csv")
            for i,(var1,var2,kno) in enumerate(lindep):
                LL12=learner.logLikelihood([var1,var2],kno)
                LL1=learner.logLikelihood([var1],kno)
                LL2=learner.logLikelihood([var2],kno)
                LL_cumul[i]+=(LL12-LL1-LL2)/size
        return [p/nbr for p in LL_cumul]

sizes=[50,100,500,1000,2000,5000,10000,20000,50000,100000,200000]
LL1,LL2,LL3,LL4,LL5,LL6=zip(*[consolidationIndepFromLL(bn,siz,
        [("visit_to_Asia","smoking",['tuberculos_or_
        ↪ cancer']),
        ("visit_to_Asia","smoking",[]),
        ("dyspnoea","smoking",[]),
        ("dyspnoea","smoking",["lung_cancer","bronchitis
        ↪ "]),
        ("tuberculosis","bronchitis",[]),
        ("tuberculosis","bronchitis",["dyspnoea"])]])
        for siz in sizes])
```

```
In [13]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
import matplotlib.patches as patches

fig=figure(figsize=(10,6))
ax = fig.add_subplot(1, 1, 1)

ax.plot(sizes,LL1,label="NOT(A indep S given TorC)", linestyle='dashed')
ax.plot(sizes,LL2,label="A indep S")
ax.plot(sizes,LL3,label="NOT(D indep S)", linestyle='dashed')
ax.plot(sizes,LL4,label="D indep S given C,B")
ax.plot(sizes,LL5,label="T indep B")
ax.plot(sizes,LL6,label="NOT(T indep B given D)", linestyle='dashed')
ax.tick_params(rotation=90)
ax.set_xlabel("data size")
ax.set_ylabel("LL/size")
ax.semilogy()
ax.legend(bbox_to_anchor=(0.15, 0.8, 0.8, .102), loc=3,ncol=3, mode="expand",_
        ↪ borderaxespad=0.)
```

(continues on next page)

(continued from previous page)

```
ax.set_title("logLikelihood=f(datasize)")
gnb.flow.add(fig)
gnb.flow.add(gnb.getBN(gum.fastBN("A->T->TorC->X;S->C->TorC->D<-B<-S")))
gnb.flow.new_line();
```



### Comparing the scores

```
In [14]: gnb.flow.display()
<IPython.core.display.HTML object>
```

```
In [ ]:
```

## 1.23 Different Graphical Models

### 1.23.1 Influence diagram

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org"> (https://agrum.org)</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html)</a>
--	--	--

```
In [1]: import os

%matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
from IPython.display import display,HTML

import math
```

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

### Build a influencediagram

#### fast build with string

```
In [3]: gum.fastID("A->*B->$C<-D<-*E->*G->H->*I<-D")
Out[3]: (pyAgrum.InfluenceDiagram<double>@00000021388516920) Influence Diagram{
    chance: 3,
    utility: 1,
    decision: 4,
    arcs: 8,
```

(continues on next page)

(continued from previous page)

```
    domainSize: 128
}
```

### bifxml format file

```
In [4]: diag=gum.loadID("res/diag.bifxml")
gnb.showInfluenceDiagram(diag)
nbsphinx-code-borderwhite
```

```
In [5]: diag
```

```
Out[5]: (pyAgrum.InfluenceDiagram<double>@00000021388516F30) Influence Diagram{
    chance: 5,
    utility: 2,
    decision: 4,
    arcs: 12,
    domainSize: 512
}
```

### the hard way :-)

```
In [6]: F=diag.addChanceNode(gum.LabelizedVariable("F","F",2))
diag.addArc(diag.idFromName("decisionVar1"),F)

U=diag.addUtilityNode(gum.LabelizedVariable("U","U",1))
diag.addArc(diag.idFromName("decisionVar3"),U)
diag.addArc(diag.idFromName("F"),U)
gnb.showInfluenceDiagram(diag)
nbsphinx-code-borderwhite
```

```
In [7]: diag.cpt(F)[{'decisionVar1':0}]=[0.9,0.1]
diag.cpt(F)[{'decisionVar1':1}]=[0.3,0.7]

diag.utility(U)[{'F':0,'decisionVar3':0}]=2
diag.utility(U)[{'F':0,'decisionVar3':1}]=4
diag.utility(U)[{'F':1}]=[0],[5]]
```

### Optimization in an influence diagram (actually LIMID)

```
In [8]: oil=gum.loadID("res/OilWildcatter.bifxml")
gnb.flow.row(oil,gnb.getInference(oil))

<IPython.core.display.HTML object>
```

```
In [9]: # a function to show results on decision nodes T and D
def show_decisions(ie):
    gnb.flow.row(ie.optimalDecision("Testing"),
                 ie.optimalDecision("Drilling"),
                 f"$$\{ie.MEU()['mean']:5.3f}\$ (stdev : \{math.sqrt(ie.MEU()[
    ↪ 'variance']):5.3f\})$$",
                 captions=["Strategy for T",
                           "Strategy for D",
```

(continues on next page)



(continued from previous page)

```

        "MEU and its standard deviation>"]
    gnb.flow.row(ie.posterior("Testing"),ie.posteriorUtility("Testing"),
        ie.posterior("Drilling"),ie.posteriorUtility("Drilling"),
        captions=["Final decision for Testing","Final reward for Testing",
            "Final decision for Drilling","Final reward for Drilling
↪"])

ie=gum.ShaferShenoyLIMIDInference(oil)

display(HTML("<h2>Inference in the LIMID optimizing the decisions nodes</h2>"))
ie.makeInference()
show_decisions(ie)
<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
<IPython.core.display.HTML object>

```

### Graphical inference with evidence and targets (developped nodes)

```

In [10]: gnb.sideBySide(oil,
    gnb.getInference(oil, evs={'TestResult': 'closed'}),
    gnb.getInference(oil, evs={'TestResult': 'open'}),
    gnb.getInference(oil, evs={'TestResult': 'diffuse'}),
    oil,
    gnb.getInference(oil, evs={'OilContents': 'Dry'}),
    gnb.getInference(oil, evs={'OilContents': 'Wet'}),
    gnb.getInference(oil, evs={'OilContents': 'Soaking'}),
    ncols=4)
<IPython.core.display.HTML object>

```

### Soft evidence on chance node

```

In [11]: gnb.showInference(oil, evs={'OilContents': [0.7, 0.5, 0.8]})
nbsphinx-code-borderwhite

```

### Forced decision

```

In [12]: gnb.showInference(oil, evs={'Drilling': 'Yes'})
nbsphinx-code-borderwhite

```

### LIMID versus Influence Diagram

The default inference for influence diagram actually an inference for LIMIDs. In order to use it for classical (and solvable) influence diagram, do not forget to add the sequence of decision nodes using `addNoForgettingAssumption`.

```

In [13]: infdiag=gum.fastID("Chance->*Decision1->Chance2->$Utility<-Chance3<-*Decision2<-
↪Chance->Utility")
infdiag

```

```
Out[13]: (pyAgrum.InfluenceDiagram<double>@00000021388517540) Influence Diagram{
    chance: 3,
    utility: 1,
    decision: 2,
    arcs: 7,
    domainSize: 32
}
```

```
In [14]: ie=gum.ShaferShenoyLIMIDInference(infdiag)
try:
    ie.makeInference()
except gum.GumException as e:
    print(e)
```

```
[pyAgrum] Fatal error: This LIMID/Influence Diagram is not solvable.
```

```
In [15]: ie.addNoForgettingAssumption(["Decision1","Decision2"])
gnb.sideBySide(ie.reducedLIMID(),ie.junctionTree(),gnb.getInference(infdiag,
↪ engine=ie))

<IPython.core.display.HTML object>
```

## Customizing visualization of the results

Using `pyAgrum.config`, it is possible to adapt the graphical representations for Influence Diagram (see [99-Tools\\_configForPyAgrum.ipynb](#)).

```
In [16]: gum.config.reset()
gnb.showInference(infdiag,engine=ie,size="7!")
nbsphinx-code-borderwhite
```

Many visual options can be changed when displaying an inference (especially for influence diagrams)

```
In [17]: # do not show inference time
gum.config["notebook","show_inference_time"]=False
# more digits for probabilities
gum.config["notebook","histogram_horizontal_visible_digits"]=3

gnb.showInference(infdiag,engine=ie,size="7!")
nbsphinx-code-borderwhite
```

```
In [18]: # specificic for influence diagram :
# more digits for utilities
gum.config["influenceDiagram","utility_visible_digits"]=5
# disabling stdev for utility and MEU
gum.config["influenceDiagram","utility_show_stdev"]=False
# showing loss (=-utility) and mEL (minimum Expected Loss) instead of MEU
gum.config["influenceDiagram","utility_show_loss"]=True

gnb.showInference(infdiag,engine=ie,size="7!")
nbsphinx-code-borderwhite
```

```
In [19]: # more visual changes for influence diagram and inference
gum.config.reset()

#shape (https://graphviz.org/doc/info/shapes.html)
gum.config["influenceDiagram","chance_shape"] = "cylinder"
```

(continues on next page)



(continued from previous page)

```
twodbn.addArc(c0,ct)
twodbn.addArc(d0,ct)
twodbn.addArc(d0,dt)

twodbn.addArc(at,ct)
twodbn.generateCPTs()

gnb.showBN(twodbn)
nbsphinx-code-borderwhite
```

## 2TBN

The dbn above actually is a 2TBN and is not correctly shown as a BN. Using the naming convention, it can be shown as a 2TBN.

```
In [3]: gdyn.showTimeSlices(twodbn)
nbsphinx-code-borderwhite
```

## unrolling 2TBN

A dbN is ‘unrolled’ using the 2TBN and the time period size. For a couple  $a_0, a_t$  in the 2TBN, the unrolled dbN will include  $a_0, a_1, \dots, a_{T-1}$

```
In [4]: T=5

dbn=gdyn.unroll2TBN(twodbn,T)
gdyn.showTimeSlices(dbn,size="10")
nbsphinx-code-borderwhite
```

We can infer on bn just as on a normal bn. Following the naming convention in 2TBN, the variables in a dbN are named using the convention  $a_i$  where  $i$  is the number of their time slice.

```
In [5]: gnb.flow.clear()
for i in range(T):
    gnb.flow.add_html(gnb.getPosterior(dbn,target="d{}".format(i),evs={}),"$P(d{})$".
    ↪format(i))
gnb.flow.display()

<IPython.core.display.HTML object>
```

## dynamic inference : following variables

`gdyn.plotFollow` directly ask for the 2TBN, unroll it and add evidence `evs`. Then it shows the dynamic of variable  $a$  for instance by plotting  $a_0, a_1, \dots, a_{T-1}$ .

```
In [6]: import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (10, 2)
gdyn.plotFollow(["a", "b", "c", "d"], twodbn, T=51, evs={'a9':2, 'a30':0, 'c14':0, 'b40':0, 'c50
    ↪':3})
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
```

## nsDBN (Non-Stationnary Dynamic Bayesian network)

In [7]:

```
T=15
dbn=gdyn.unroll2TBN(twodbn,T)
gdyn.showTimeSlices(dbn)
nbsphinx-code-borderwhite
```

Non-stationnary DBN allows to express that the dBN do not follow the same 2TBN during all steps. A unrolled dbn is a classical BayesNet and then can be changed as you want after unrolling.

In [8]:

```
# new P(ct/c0)
pot=gum.Potential().add(twodbn.variableFromName("ct")).add(twodbn.variableFromName("c0
↪"))
pot.fillWith([1,0,0,0.1]*9).normalizeAsCPT() # 36 valeurs normalized as CPT
```

Out[8]:

```
(pyAgrum.Potential<double>@0000001D6E95E2280)
|| ct
c0 || 0 | 1 | 2 | 3 | 4 | 5 |
-----|-----|-----|-----|-----|-----|-----|
0 || 0.4762 | 0.0000 | 0.0000 | 0.0476 | 0.4762 | 0.0000 |
1 || 0.0000 | 0.0833 | 0.8333 | 0.0000 | 0.0000 | 0.0833 |
2 || 0.4762 | 0.0000 | 0.0000 | 0.0476 | 0.4762 | 0.0000 |
3 || 0.0000 | 0.0833 | 0.8333 | 0.0000 | 0.0000 | 0.0833 |
4 || 0.4762 | 0.0000 | 0.0000 | 0.0476 | 0.4762 | 0.0000 |
5 || 0.0000 | 0.0833 | 0.8333 | 0.0000 | 0.0000 | 0.0833 |
```

In [9]:

```
# from steps 5 to 10, $C_t$ only depends on $C_{t-1}$ and follows this new CPT
for i in range(5,11):
    dbn.eraseArc(f"d{i-1}",f"c{i}")
    dbn.eraseArc(f"a{i}",f"c{i}")
    dbn.cpt(f"c{i}").fillWith(pot,["ct","c0"]) # ct in pot <- first var of cpt, c0 in
↪pot<-second var in cpt

gdyn.showTimeSlices(dbn,size="14")
nbsphinx-code-borderwhite
```

In [10]:

```
plt.rcParams['figure.figsize'] = (10, 2)
gdyn.plotFollowUnrolled(["a","b","c","d"],dbn,T=15,evs={'a9':2,'c14':0})
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
```

In [ ]:

### 1.23.3 Markov networks



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: %load_ext autoreload
        %autoreload 2

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.mn2graph as m2g
```

### building a Markov Network

```
In [2]: gum.config.reset() # back to default
mn=gum.fastMN("A--B--C;C--D;B--E--F;F--D--G;H--J;E--A;J")
mn
```

```
Out[2]: (pyAgrum.MarkovNet<double>@000000269F09E9870) MN{nodes: 9, edges: 12, domainSize: 512,
↪dim: 38}
```

Using `pyAgrum.config`, it is possible to adapt the graphical representations for Markov Network (see [99-Tools\\_configForPyAgrum.ipynb](#)).

```
In [3]: gum.config.reset() # back to default
gum.config['factorgraph','edge_length']='0.4'
gnb.showMN(mn)
nbsphinx-code-borderwhite
```

```
In [4]: gum.config.reset() # back to default
print("Default view for Markov network: "+gum.config['notebook','default_
↪markovnetwork_view'])
gum.config['notebook','default_markovnetwork_view']='graph'
print("modified to: "+gum.config['notebook','default_markovnetwork_view'])
mn
```

```
Default view for Markov network: factorgraph
modified to: graph
```

```
Out[4]: (pyAgrum.MarkovNet<double>@000000269F09E9870) MN{nodes: 9, edges: 12, domainSize: 512,
↪dim: 38}
```

```
In [5]: gnb.sideBySide(gnb.getMN(mn,view="graph",size="5"),
                      gnb.getMN(mn,view="factorgraph",size="5"))

<IPython.core.display.HTML object>
```

```
In [6]: gnb.showMN(mn)
print(mn)
nbsphinx-code-borderwhite
MN{nodes: 9, edges: 12, domainSize: 512, dim: 38}
```

## Accessors for Markov Networks

```
In [7]: print(f"nodes      : {mn.nodes()}")
print(f"node names  : {mn.names()}")
print(f"edges       : {mn.edges()}")
print(f"components  : {mn.connectedComponents()}")
print(f"factors      : {mn.factors()}")
print(f"factor(C,D) : {mn.factor({2,3})}")
print(f"factor(C,D) : {mn.factor({'C','D'})}")
print(f"factor(C,D) : {mn.factor({'D','C'})}")
```

```
nodes      : {0, 1, 2, 3, 4, 5, 6, 7, 8}
node names  : {'B', 'J', 'D', 'G', 'C', 'A', 'F', 'E', 'H'}
edges       : {(0, 1), (1, 2), (0, 4), (1, 5), (1, 4), (2, 3), (4, 5), (0, 2), (5, 6),
↪ (7, 8), (3, 6), (3, 5)}
components  : {0: {0, 1, 2, 3, 4, 5, 6}, 7: {8, 7}}
factors      : [{0, 1, 2}, {2, 3}, {8, 7}, {1, 4, 5}, {3, 5, 6}, {0, 4}, {8}]
factor(C,D) :
```

		C	
D		0	1
0		0.9965	0.9677
1		0.7258	0.9811

```
factor(C,D) :
```

		C	
D		0	1
0		0.9965	0.9677
1		0.7258	0.9811

```
factor(C,D) :
```

		C	
D		0	1
0		0.9965	0.9677
1		0.7258	0.9811

```
In [8]: try:
        mn.factor({0,1})
except gum.GumException as e:
    print(e)
try:
    mn.factor({"A","B"})
except gum.GumException as e:
    print(e)
```

```
[pyAgrum] Object not found: No element with the key <{1,0}>
[pyAgrum] Object not found: No element with the key <{1,0}>
```

## Manipulating factors

In [9]: `mn.factor({'A','B','C'})`

Out[9]: `(pyAgrum.Potential<double>@000000269CD3AAAE0)`

		A	
B	C	0	1
0	0	0.1355	0.8350
1	0	0.9689	0.2210
0	1	0.3082	0.5472
1	1	0.1884	0.9929

In [10]: `mn.factor({'A','B','C'})[{'B':0}]`

Out[10]: `array([[0.135477, 0.83500859],  
[0.30816705, 0.5472206 ]])`

In [11]: `mn.factor({'A','B','C'})[{'B':0}]=[1,2],[3,4]  
mn.factor({'A','B','C'})`

Out[11]: `(pyAgrum.Potential<double>@000000269CD3AAAE0)`

		A	
B	C	0	1
0	0	1.0000	2.0000
1	0	0.9689	0.2210
0	1	3.0000	4.0000
1	1	0.1884	0.9929

## Customizing graphical representation

In [12]: `gum.config.reset() # back to default  
gum.config['factorgraph','edge_length']='0.5'`

```
maxnei=max([len(mn.neighbours(n)) for n in mn.nodes()])
nodemap={n:len(mn.neighbours(mn.idFromName(n)))/maxnei for n in mn.names()}

facmax=max([len(f) for f in mn.factors()])
fgma=lambda factor: (1+len(factor)**2)/(1+facmax*facmax)

gnb.flow.row(gnb.getGraph(m2g.MN2UGdot(mn)),
              gnb.getGraph(m2g.MN2UGdot(mn,nodeColor=nodemap)),
              gnb.getGraph(m2g.MN2FactorGraphdot(mn)),
              gnb.getGraph(m2g.MN2FactorGraphdot(mn,factorColor=fgma,
->nodeColor=nodemap))),
              captions=['Markov network',
                        'MarkovNet with colored node w.r.t number of neighbours',
                        'Markovnet as factor graph',
                        'MN with colored factor w.r.t to the size of scope'])

<IPython.core.display.HTML object>
```



### from BayesNet to MarkovNet

```
In [13]: bn=gum.fastBN("A->B<-C->D->E->F<-B<-G;A->H->I;C->J<-K<-L")
mn=gum.MarkovNet.fromBN(bn)
gnb.flow.row(bn,
              gnb.getGraph(m2g.MN2UGdot(mn)),
              captions=['a Bayesian network',
                       'the corresponding Markov Network'])

<IPython.core.display.HTML object>
```

```
In [14]: # The corresponding factor graph
m2g.MN2FactorGraphdot(mn)
```

```
Out[14]: <pydot.Dot at 0x269f2cc5ea0>
```

### Inference in Markov network

```
In [15]: bn=gum.fastBN("A->B<-C->D->E->F<-B<-G;A->H->I;C->J<-K<-L")
iebn=gum.LazyPropagation(bn)

mn=gum.MarkovNet.fromBN(bn)
iemn=gum.ShaferShenoyMNIInference(mn)
iemn.setEvidence({"A":1,"F":[0.4,0.8]})
iemn.makeInference()
iemn.posterior("B")
```

```
Out[15]: (pyAgrum.Potential<double>@00000269F0EFA580)
      B      |
0      | 1      |
-----|-----|
0.6152 | 0.3848 |
```

```
In [16]: def affAGC(evs):
          gnb.sideBySide(gnb.getSideBySide(gum.getPosterior(bn,target="A",evs=evs),
                                              gum.getPosterior(bn,target="G",evs=evs),
                                              gum.getPosterior(bn,target="C",evs=evs)),
                        gnb.getSideBySide(gum.getPosterior(mn,target="A",evs=evs),
                                              gum.getPosterior(mn,target="G",evs=evs),
                                              gum.getPosterior(mn,target="C",evs=evs)),
          captions=["Inference in the Bayesian network bn with evidence
↪"+str(evs),
                  "Inference in the markov network mn with evidence
↪"+str(evs)]
          )

print("Inference for both the corresponding models in BayesNet and MarkovNet worlds.
↪when the MN comes from a BN")
affAGC({})
print("C has no impact on A and G")
affAGC({'C':1})

print("But if B is observed")
affAGC({'B':1})
print("C has an impact on A and G")
affAGC({'B':1,'C':0})
```

Inference for both the corresponding models in BayesNet and MarkovNet worlds when the `↪MN` comes from a BN

<IPython.core.display.HTML object>

C has no impact on A and G

<IPython.core.display.HTML object>

But if B is observed

<IPython.core.display.HTML object>

C has an impact on A and G

<IPython.core.display.HTML object>

```
In [17]: mn.generateFactors()
print("But with more general factors")
affAGC({})
print("C has impact on A and G even without knowing B")
affAGC({'C':1})
```

But with more general factors

<IPython.core.display.HTML object>

C has impact on A and G even without knowing B

<IPython.core.display.HTML object>

### Graphical inference in markov network

```
In [18]: bn=gum.fastBN("A->B<-C->D->E->F<-B<-G;A->H->I;C->J<-K<-L")
mn=gum.MarkovNet.fromBN(bn)

gnb.sideBySide(gnb.getJunctionTree(bn),gnb.getJunctionTree(mn),captions=["Junction
↪tree for the BN","Junction tree for the induced MN"])
gnb.sideBySide(gnb.getJunctionTreeMap(bn,size="3!"),gnb.getJunctionTreeMap(mn,size="3!
↪"),captions=["Map of the junction tree for the BN","Map of the junction tree for
↪the induced MN"])

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
```

```
In [19]: gnb.showInference(bn, evs={"D":1, "H":0})
nbsphinx-code-borderwhite
```

```
In [20]: gum.config.reset()
gnb.showInference(mn, size="8", evs={"D":1, "H":0})
nbsphinx-code-borderwhite
```



```
In [21]: gum.config['factorgraph', 'edge_length_inference']='1.1'
gnb.showInference(mn, size="11", evs={"D":1, "H":0})
nbsphinx-code-borderwhite
```

```
In [22]: gum.config['notebook', 'default_markovnetwork_view']='graph'
gnb.showInference(mn, size="8", evs={"D":1, "H":0})
```

nbsphinx-code-borderwhite

In [ ]:

### 1.23.4 Credal Networks

 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrums.org">https://agrums.org</a>)</p>	<p>(<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html">https://agrums.gitlab.io/extra/agrum_at_binder.html</a>)</p>
--	--	--

In [1]: `import os`

```
%matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
```

In [2]: `import pyAgrum as gum`  
`import pyAgrum.lib.notebook as gnb`  
`gnb.configuration()`

<IPython.core.display.HTML object>

#### Credal Net from BN

```
In [3]: bn=gum.fastBN("A->B[3]->C<-D<-A->E->F")
bn_min=gum.BayesNet(bn)
bn_max=gum.BayesNet(bn)
for n in bn.nodes():
    x=0.4*min(bn.cpt(n).min(),1-bn.cpt(n).max())
    bn_min.cpt(n).translate(-x)
    bn_max.cpt(n).translate(x)

cn=gum.CredalNet(bn_min,bn_max)
cn.intervalToCredal()

gnb.flow.row(bn,bn.cpt("B"),cn,bn_min.cpt("B"),bn_max.cpt("B"),captions=["Bayes Net",
↪ "CPT", "Credal Net", "CPTmin", "CPTmax"])
```

<IPython.core.display.HTML object>

We can use LBP on CN (L2U) only for binary credal networks (here  $B$  is not binary). We then propose the classical binarization (but warn the user that this leads to approximation in the inference)

```
In [4]: cn2=gum.CredalNet(bn_min,bn_max)
cn2.intervalToCredal()
cn2.approximatedBinarization()
cn2.computeBinaryCPTMinMax()

gnb.flow.row(cn,cn2,captions=["Credal net","Binarized credal net"])

<IPython.core.display.HTML object>
```

Here,  $B$  becomes -  $B$ - $i$  : the  $i$ -th bit of  $B$  - instrumental  $B$ - $k$  : the indicator variable for each modality  $k$  of  $B$

```
In [5]: ie_mc=gum.CNMonteCarloSampling(cn)
ie2_lbp=gum.CNLoopyPropagation(cn2)
ie2_mc=gum.CNMonteCarloSampling(cn2)
```

```
In [6]: gnb.sideBySide(gnb.getInference(cn,engine=ie_mc),
                    gnb.getInference(cn2,engine=ie2_mc),
                    gnb.getInference(cn2,engine=ie2_lbp))

<IPython.core.display.HTML object>
```

```
In [7]: gnb.sideBySide(ie_mc.CN(),ie_mc.marginalMin("F"),ie_mc.marginalMax("F"),
                    ie_mc.CN(),ie2_lbp.marginalMin("F"),ie2_lbp.marginalMax("F"),
                    ncols=3)

print(cn)

<IPython.core.display.HTML object>
```

```
A:Range([0,1])
<> : [[0.0837589 , 0.916241] , [0.195435 , 0.804565]]

B:Range([0,2])
<A:0> : [[0.60315 , 0.147547 , 0.249304] , [0.60315 , 0.19114 , 0.20571] , [0.646745 ,
→ 0.19114 , 0.162115] , [0.690339 , 0.147547 , 0.162115] , [0.646744 , 0.103952 , 0.
→ 249304] , [0.690339 , 0.103952 , 0.20571]]
<A:1> : [[0.272995 , 0.108987 , 0.618018] , [0.272995 , 0.152582 , 0.574423] , [0.
→ 316589 , 0.152582 , 0.530829] , [0.360184 , 0.108987 , 0.530829] , [0.31659 , 0.
→ 0653924 , 0.618018] , [0.360184 , 0.0653924 , 0.574424]]

C:Range([0,1])
<B:0|D:0> : [[0.500982 , 0.499018] , [0.513663 , 0.486337]]
<B:1|D:0> : [[0.418886 , 0.581114] , [0.431566 , 0.568434]]
<B:2|D:0> : [[0.114657 , 0.885343] , [0.127336 , 0.872664]]
<B:0|D:1> : [[0.977811 , 0.0221893] , [0.990491 , 0.00950872]]
<B:1|D:1> : [[0.143169 , 0.856831] , [0.155849 , 0.844151]]
<B:2|D:1> : [[0.62924 , 0.37076] , [0.641919 , 0.358081]]

D:Range([0,1])
<A:0> : [[0.593434 , 0.406566] , [0.783228 , 0.216772]]
<A:1> : [[0.142344 , 0.857656] , [0.332139 , 0.667861]]

E:Range([0,1])
```

(continues on next page)

(continued from previous page)

```

<A:0> : [[0.209976 , 0.790024] , [0.489943 , 0.510057]]
<A:1> : [[0.389381 , 0.610619] , [0.669348 , 0.330652]]

F:Range([0,1])
<E:0> : [[0.251754 , 0.748246] , [0.479112 , 0.520888]]
<E:1> : [[0.602119 , 0.397881] , [0.82948 , 0.17052]]

```

### Credal Net from bif files

```
In [8]: cn=gum.CredalNet("res/cn/2Umin.bif","res/cn/2Umax.bif")
cn.intervalToCredal()
```

```
In [9]: gnb.showCN(cn,"2")
nbsphinx-code-borderwhite
```

```
In [10]: ie=gum.CNMonteCarloSampling(cn)
ie.insertEvidenceFile("res/cn/L2U.evi")
```

```
In [11]: ie.setRepetitiveInd(False)
ie.setMaxTime(1)
ie.setMaxIter(1000)

ie.makeInference()
```

```
In [12]: cn
```

```
Out[12]: (pyAgrum.CredalNet<double>@0000001F6ED4EF310)
A:Labelized({0|1})
<> : [[0.6 , 0.4] , [0.7 , 0.3]]

B:Labelized({0|1})
<> : [[0.6 , 0.4] , [0.8 , 0.2]]

C:Labelized({0|1})
<> : [[0 , 1] , [0.1 , 0.9]]

D:Labelized({0|1})
<> : [[0.1 , 0.9] , [0.5 , 0.5]]

E:Labelized({0|1})
<A:0|B:0> : [[0.3 , 0.7] , [0.4 , 0.6]]
<A:1|B:0> : [[0.8 , 0.2] , [1 , 0]]
<A:0|B:1> : [[0.7 , 0.3] , [0.9 , 0.1]]
<A:1|B:1> : [[0.5 , 0.5] , [0.7 , 0.3]]

F:Labelized({0|1})
<C:0|D:0> : [[0.1 , 0.9] , [0.2 , 0.8]]
<C:1|D:0> : [[0.5 , 0.5]]
<C:0|D:1> : [[0.3 , 0.7] , [0.5 , 0.5]]
<C:1|D:1> : [[0.6 , 0.4] , [0.9 , 0.1]]

G:Labelized({0|1})
```

(continues on next page)

(continued from previous page)

```
<D:0> : [[0.6 , 0.4] , [0.8 , 0.2]]
<D:1> : [[0.2 , 0.8] , [0.3 , 0.7]]

H:Labelized({0|1})
<E:0|F:0> : [[0 , 1] , [0.1 , 0.9]]
<E:1|F:0> : [[0.6 , 0.4] , [0.8 , 0.2]]
<E:0|F:1> : [[0.2 , 0.8] , [0.4 , 0.6]]
<E:1|F:1> : [[0.8 , 0.2] , [0.9 , 0.1]]

L:Labelized({0|1})
<H:0> : [[0.8 , 0.2] , [1 , 0]]
<H:1> : [[0.5 , 0.5] , [0.6 , 0.4]]
```

```
In [13]: gnb.showInference(cn,targets={"A","H","L","D"},engine=ie,evs={"L":[0,1],"G":[1,0]})
nbsphinx-code-borderwhite
```

### Comparing inference in credal networks

```
In [14]: import pyAgrum as gum

def showDiffInference(model,mc,lbp):
    for i in model.current_bn().nodes():
        a,b=mc.marginalMin(i)[:]
        c,d=mc.marginalMax(i)[:]

        e,f=lbp.marginalMin(i)[:]
        g,h=lbp.marginalMax(i)[:]

        plt.scatter([a,b,c,d],[e,f,g,h])

cn=gum.CredalNet("res/cn/2Umin.bif","res/cn/2Umax.bif")
cn.intervalToCredal()
```

### The two inference give quite the same result

```
In [15]: ie_mc=gum.CNMonteCarloSampling(cn)
ie_mc.makeInference()

cn.computeBinaryCPTMinMax()
ie_lbp=gum.CNLoopyPropagation(cn)
ie_lbp.makeInference()

showDiffInference(cn,ie_mc,ie_lbp)
nbsphinx-code-borderwhite
```

**but not when evidence are inserted**

```
In [16]: ie_mc=gum.CNMonteCarloSampling(cn)
         ie_mc.insertEvidenceFile("res/cn/L2U.evi")
         ie_mc.makeInference()

         ie_lbp=gum.CNLoopyPropagation(cn)
         ie_lbp.insertEvidenceFile("res/cn/L2U.evi")
         ie_lbp.makeInference()

         showDiffInference(cn,ie_mc,ie_lbp)
```

nbsphinx-code-borderwhite

**Dynamical Credal Net**

```
In [17]: cn=gum.CredalNet("res/cn/bn_c_8.bif","res/cn/den_c_8.bif")
         cn.bnToCredal(0.8,False)
```

```
In [18]: ie=gum.CNMonteCarloSampling(cn)
         ie.insertModalsFile("res/cn/modalities.modal")

         ie.setRepetitiveInd(True)
         ie.setMaxTime(5)
         ie.setMaxIter(1000)

         ie.makeInference()
```

```
In [19]: print(ie.dynamicExpMax("temp"))

(14.20340464862347, 11.769513762315974, 12.190483075680442, 12.017742181033096, 12.
↪ 004198572496797, 12.008328195599265, 12.007694250617146, 12.007688078235907, 12.
↪ 00772208700393)
```

```
In [20]: fig=figure()
         ax=fig.add_subplot(111)
         ax.fill_between(range(9),ie.dynamicExpMax("temp"),ie.dynamicExpMin("temp"))
```

```
Out[20]: <matplotlib.collections.PolyCollection at 0x1f6f4262890>
```

nbsphinx-code-borderwhite

```
In [21]: ie=gum.CNMonteCarloSampling(cn)
         ie.insertModalsFile("res/cn/modalities.modal")

         ie.setRepetitiveInd(False)
         ie.setMaxTime(5)
         ie.setMaxIter(1000)

         ie.makeInference()
         print(ie.messageApproximationScheme())

         stopped with epsilon=0
```

```
In [22]: fig=figure()
         ax=fig.add_subplot(111)
         ax.fill_between(range(9),ie.dynamicExpMax("temp"),ie.dynamicExpMin("temp"))
```

```
Out[22]: <matplotlib.collections.PolyCollection at 0x1f6f415e170>
nbsphinx-code-borderwhite
```

```
In [23]: ie=gum.CNMonteCarloSampling(cn)
ie.insertModalsFile("res/cn/modalities.modal")



ie.setRepetitiveInd(False)
ie.setMaxTime(5)
ie.setMaxIter(5000)

gnb.animApproximationScheme(ie)
ie.makeInference()
nbsphinx-code-borderwhite
```

```
In [24]: fig=figure()
ax=fig.add_subplot(111)
ax.fill_between(range(9),ie.dynamicExpMax("temp"),ie.dynamicExpMin("temp"));
nbsphinx-code-borderwhite
```

```
In [ ]:
```

### 1.23.5 Object-Oriented Probabilistic Relational Model

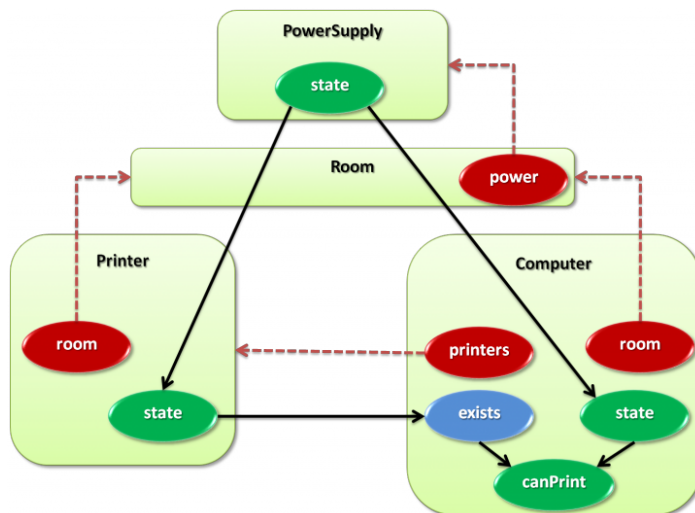
 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrum.org">https://agrum.org</a>)</p>	<p>(<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>)</p>
--	--	--

```
In [1]: import os
os.chdir("res")

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

O3PRM is inspired by relational language, directly based on the BN and improved by object oriented paradigms, where the focus is set on classes of objects and by defining relations among these objects (see <http://o3prm.gitlab.io/> for more details)





### O3PRM syntax

```
In [2]: with open('./Asia.o3prm','r') as prm:
        for line in prm.readlines():
            print(line,end="")
```

```
class Asia {

    boolean visitToAsia {
        [ 0.99, // False
          0.01 ] // True
    };

    boolean tuberculosis dependson visitToAsia {
        // False | True
        [   0.99, 0.95, // False
          0.01, 0.05 ] // True
    };

    boolean smoking {
        [ 0.50, // False
          0.50 ] // True
    };

    boolean lungCancer dependson smoking {
        // False | True => smoking
        [   0.99, 0.90, // False
          0.01, 0.10 ] // True
    };

    boolean bronchitis dependson smoking {
        // False | True => smoking
        [   0.70, 0.40, // False
          0.30, 0.60 ] // True
    };

    boolean tubOrCancer dependson tuberculosis, lungCancer {
        //      False  ||      True      => tuberculosis
        // False | True || False | True  => lungCancer
        [   1.00, 0.00,   0.00, 0.00, // False
```

(continues on next page)

(continued from previous page)

```

        0.00, 1.00,      1.00, 1.00 ] // True
};

boolean positiveXRay dependson tubOrCancer {
    // False | True => tubOrCancer
    [ 0.95, 0.02, // False
      0.05, 0.98 ] // True
};

boolean dyspnea dependson tubOrCancer, bronchitis {
    //      False    ||      True      => tubOrCancer
    // False | True || False | True => bronchitis
    [   0.90, 0.20,   0.30, 0.10, // False
      0.10, 0.80,   0.70, 0.90 ] // True
};
}

```

### Using o3prm syntax for creating BayesNet

```
In [3]: bn=gum.loadBN("./Asia.o3prm",verbose=False)
bn
```

```
Out[3]: (pyAgrum.BayesNet<double>@0000001C0FD7C9590) BN{nodes: 8, arcs: 8, domainSize: 256,
↳ dim: 36}
```

```
In [4]: bn=gum.loadBN("./aSys.o3prm")
bn
```

```
Out[4]: (pyAgrum.BayesNet<double>@0000001C0FD7CB020) BN{nodes: 10, arcs: 9, domainSize: 1024,
↳ dim: 38}
```

```
In [5]: classpath="./ComplexPrinters"
filename="./ComplexPrinters/fr/lip6/printers/system.o3prm"

system="Work"
bn=gum.loadBN(filename,system=system,classpath=classpath)
```

```
In [6]: # the inference will take place in a rather large junction tree
gnb.showJunctionTreeMap(bn,scaleClique=0.1,scaleSep=0.05,lenEdge=1.0,size="8!")
nbsphinx-code-borderwhite
```

```
In [7]: gnb.showInference(bn,size="25!")
nbsphinx-code-borderwhite
```

```
In [8]: # if cairosVG is installed, one can export a graph (or inference) as a pdf
# gnb.exportInference(bn,filename="../out/ComplexPrinters.pdf")
```

```
In [9]: bn=gum.loadBN("./aSys.o3prm")
gnb.sideBySide(
    gnb.getBN(bn,size='5'),
    gnb.getInference(bn,size='5')
)
```

```
<IPython.core.display.HTML object>
```

## Exploring Probabilistic Relational Model

```
In [10]: classpath="./ComplexPrinters"
filename="./ComplexPrinters/complexprinters_system.o3prm"
explor=gum.PRMexplorer()
explor.load(filename)
```

```
In [11]: for cl in explor.classes():
    print("Class : "+cl)
    print(" - Super class : "+ ("None" if explor.getSuperClass(cl)==None else explor.
    ↪getSuperClass(cl)) )
    print(" - Implemented interface : ")
    for inter in explor.classImplements(cl):
        print(" "+inter)
    print(" - Direct sub-types : ")
    for ext in explor.getDirectSubClass(cl):
        print(" "+ext)
    print(" - Attributes : ")
    for (t,n, depensons) in explor.classAttributes(cl):
        s = ""
        for depenson in depensons:
            s = s + depenson + " "
        print(" "+t+" "+n+" (" +s+")")
    print(" - References : ")
    for (t, n, isArray) in explor.classReferences(cl):
        print(" "+t+"([" if isArray else "")+" "+n)
    print(" - Aggragates : ")
    for (t, n, g, l, slots) in explor.classAggregates(cl):
        s = ""
        for slot in slots:
            s = s + slot + " "
        print(" "+t+" "+n+" "+g+" "+("NoLabel" if l==None else l)+ " (" +s+")")
    print(" - SlotChains : ")
    for (t, n, isMultiple) in explor.classSlotChains(cl):
        print(" "+t+" "+n+" "+("[" if isMultiple else ""))
    print(" - Parameters : ")
    for param in explor.classParameters(cl):
        print(" "+param)
    #print(" - Dag : ")
    #(dic, dotString) = explor.classDag(cl)
    #print(dic)
    #print(dotString)

    print()
```

```
Class : Room
- Super class : None
- Implemented interface :
- Direct sub-types :
- Attributes :
- References :
    PowerSupply power
- Aggragates :
- SlotChains :
- Parameters :
```

```
Class : Computer
```

(continues on next page)

(continued from previous page)

- Super class : None
- Implemented interface :  
Equipment
- Direct sub-types :
- Attributes :  
t\_degraded equipState (room.power.state )  
boolean can\_print (equipState working\_printer )
- References :  
Printer[] printers  
Room room
- Aggragates :  
boolean functional\_printer exists OK (printers.equipState )  
boolean working\_printer exists true (degraded\_printer functional\_printer )  
boolean degraded\_printer exists Degraded (printers.equipState )  
mycount count\_printers count true (degraded\_printer functional\_printer )
- SlotChains :  
t\_degraded printers.equipState []  
t\_state room.power.state
- Parameters :

Class : SafeComputer

- Super class : None
- Implemented interface :  
Equipment
- Direct sub-types :
- Attributes :  
t\_degraded equipState (room.power.(t\_state)state )  
boolean can\_print (working\_printer equipState )
- References :  
Room room  
Printer[] printers
- Aggragates :  
boolean functional\_printer exists OK (printers.(t\_state)equipState )  
boolean degraded\_printer exists Degraded (printers.(t\_degraded)equipState )  
boolean working\_printer exists true (functional\_printer degraded\_printer )
- SlotChains :  
t\_state printers.(t\_state)equipState []  
t\_degraded printers.(t\_degraded)equipState []  
t\_state room.power.(t\_state)state
- Parameters :

Class : ParamClass

- Super class : None
- Implemented interface :
- Direct sub-types :  
ParamClass<lambda=0.001,t=4>  
ParamClass<lambda=0.4,t=4>
- Attributes :  
t\_degraded equipState (room.power.state hasInk hasPaper )  
t\_ink hasInk ()  
t\_paper hasPaper ()
- References :  
Room room
- Aggragates :
- SlotChains :  
t\_state room.power.state

(continues on next page)

(continued from previous page)

- Parameters :
- lambda
- t

Class : ParamClass<lambda=0.001,t=4>

- Super class : ParamClass
- Implemented interface :
- Direct sub-types :
- Attributes :
  - t\_degraded equipState (room.power.state hasInk hasPaper )
  - t\_paper hasPaper ()
  - t\_ink hasInk ()
- References :
  - Room room
- Aggragates :
- SlotChains :
  - t\_state room.power.state
- Parameters :
  - lambda
  - t

Class : PowerSupply

- Super class : None
- Implemented interface :
- Direct sub-types :
- Attributes :
  - t\_state state ()
- References :
- Aggragates :
- SlotChains :
- Parameters :

Class : BWPrinter

- Super class : None
- Implemented interface :
  - Printer
- Direct sub-types :
- Attributes :
  - t\_ink hasInk ()
  - t\_paper hasPaper ()
  - t\_degraded equipState (room.power.state hasPaper hasInk )
- References :
  - Room room
- Aggragates :
- SlotChains :
  - t\_state room.power.state
- Parameters :

Class : ColorPrinter

- Super class : None
- Implemented interface :
  - Printer
- Direct sub-types :
- Attributes :
  - t\_ink yellow ()
  - t\_ink magenta ()

(continues on next page)

(continued from previous page)

```

    t_paper hasPaper ()
    t_degraded equipState (room.power.state hasPaper hasInk black )
    t_ink cyan ()
    t_ink black ()
- References :
    Room room
- Aggregates :
    boolean hasInk forall NotEmpty (yellow cyan black magenta )
- SlotChains :
    t_state room.power.state
- Parameters :

Class : ParamClass<lambda=0.4,t=4>
- Super class : ParamClass
- Implemented interface :
- Direct sub-types :
- Attributes :
    t_ink hasInk ()
    t_degraded equipState (room.power.state hasInk hasPaper )
    t_paper hasPaper ()
- References :
    Room room
- Aggregates :
- SlotChains :
    t_state room.power.state
- Parameters :
    t
    lambda

```

```

In [12]: print("The following lists the systems of the prm:\n")
systems=explor.getalltheSystems()
sys1=systems[0]
print("Name of the system: "+sys1[0]+"\n")
print("Nodes : dict(id: [name,type])")
print(sys1[1])
print("\n")
print("Arcs : List[(tail, head),(tail, head)...]")
print(sys1[2])

```

The following lists the systems of the prm:

Name of the system: aSys

Nodes : dict(id: [name,type])

```

{0: ('pow', 'PowerSupply'), 1: ('r', 'Room'), 2: ('bw_printers[0]', 'BWPrinter'), 3: (
→ 'bw_printers[1]', 'BWPrinter'), 4: ('bw_printers[2]', 'BWPrinter'), 5: ('bw_
→ printers[3]', 'BWPrinter'), 6: ('bw_printers[4]', 'BWPrinter'), 7: ('bw_printers[5]
→ ', 'BWPrinter'), 8: ('bw_printers[6]', 'BWPrinter'), 9: ('bw_printers[7]',
→ 'BWPrinter'), 10: ('bw_printers[8]', 'BWPrinter'), 11: ('bw_printers[9]', 'BWPrinter
→ '), 12: ('color_printers[0]', 'ColorPrinter'), 13: ('color_printers[1]',
→ 'ColorPrinter'), 14: ('c1', 'Computer'), 15: ('c2', 'Computer'), 16: ('p',
→ 'ParamClass<lambda=0.4,t=4>'), 17: ('paramBis', 'ParamClass<lambda=0.001,t=4>')}

```

Arcs : List[(tail, head),(tail, head)...]

[]

```
In [13]: gnb.showPotential(explor.cpf('Computer','equipState'))
```

```
<IPython.core.display.HTML object>
```

```
In [14]: for cl in explor.types():
    print("Type : "+cl)
    print(" - Super type : "+ ("None" if explor.getSuperType(cl)==None else explor.
    ↪getSuperType(cl)) )
    print(" - Direct sub-types : ")
    for name in explor.getDirectSubTypes(cl):
        print(" "+name)
    print(" - Labels : ")
    for t in explor.getLabels(cl):
        print(" "+t)
    print(" - Labels mapping : ")
    for key,val in dict().items() if explor.getLabelMap(cl)==None else explor.
    ↪getLabelMap(cl).items():
        print(" "+key+ ' -> '+val)
    print()
```

```
Type : t_ink
- Super type : t_state-000001C0888DA180
- Direct sub-types :
- Labels :
    NotEmpty
    Empty
- Labels mapping :
    NotEmpty -> OK
    Empty -> NOK
```

```
Type : boolean
- Super type : None
- Direct sub-types :
    t_state
- Labels :
    false
    true
- Labels mapping :
```

```
Type : t_degraded
- Super type : t_state-000001C0888DA180
- Direct sub-types :
- Labels :
    OK
    Dysfunctional
    Degraded
- Labels mapping :
    OK -> OK
    Dysfunctional -> NOK
    Degraded -> NOK
```

```
Type : t_state
- Super type : boolean-000001C0888D95A0
- Direct sub-types :
    t_ink
    t_degraded
```

(continues on next page)

(continued from previous page)

```

    t_paper
- Labels :
    OK
    NOK
- Labels mapping :
    OK -> true
    NOK -> false

```

Type : t\_paper

```

- Super type : t_state-000001C0888DA180
- Direct sub-types :
- Labels :
    Ready
    Jammed
    Empty
- Labels mapping :
    Ready -> OK
    Jammed -> NOK
    Empty -> NOK

```

Type : mycount

```

- Super type : None
- Direct sub-types :
- Labels :
    0
    1
    2
    3
    4
    5
- Labels mapping :

```

```

In [15]: for cl in explor.interfaces():
    print("Interface : "+cl)
    print(" - Super interface : "+ ("None" if explor.getSuperInterface(cl)==None
    ↪ else explor.getSuperInterface(cl)) )
    print(" - Direct sub-interfaces : ")
    for name in explor.getDirectSubInterfaces(cl):
        print("         "+name)
    print(" - Implementations :")
    for impl in explor.getImplementations(cl):
        print("         "+impl)
    print(" - Attributes : ")
    for (t,n) in explor.interAttributes(cl, allAttributes=True):
        print("         "+t+" "+n)
    print(" - References : ")
    for (t, n, isArray) in explor.interReferences(cl):
        print("         "+t+("[]" if isArray else "")+" "+n)
    print()

```

Interface : Printer

```

- Super interface : Equipment
- Direct sub-interfaces :
- Implementations :
    BWPrinter

```

(continues on next page)





(continued from previous page)

```
import os
import pandas
```

skbn is a pyAgrum's module that allows to use bayesian networks as classifier in the scikit-learn environment. ## Initialization of parameters

First, we initialize the parameters to indicate properties we want our classifier to have.

```
In [2]: BNTest= skbn.BNClassifier(learningMethod = 'Chow-Liu', prior= 'Smoothing',
    ↪priorWeight = 0.5,
    ↪discretizationStrategy = 'quantile', usePR = True,
    ↪significant_digit = 13)
```

Then, we train the classifier thanks to two types of objects.

### Learn from csv file

```
In [3]: BNTest.fit(filename = 'res/creditCardTest.csv', targetName = 'Class')

**pyAgrum** : 'filename' is deprecated since 1.1.1. Please use 'data' instead.
```

```
In [4]: for i in BNTest.bn.nodes():
    ↪print(BNTest.bn.variable(i))

Class:Labelized({0.0|1.0})
Time:Discretized(<(-0;1578[, [1578;3733[, [3733;6982[, [6982;11033[, [11033;170348]>>)
V1:Discretized(<(-30.5524;-1.33295[, [-1.33295;-0.654664[, [-0.654664;0.305375[, [0.
    ↪305375;1.18346[, [1.18346;2.13239]>>)
V2:Discretized(<(-25.6405;-0.362408[, [-0.362408;0.104022[, [0.104022;0.582468[, [0.
    ↪582468;1.12626[, [1.12626;22.0577]>>)
V3:Discretized(<(-31.1037;0.107723[, [0.107723;0.675277[, [0.675277;1.14525[, [1.14525;1.
    ↪73106[, [1.73106;4.10172]>>)
V4:Discretized(<(-4.65755;-0.835683[, [-0.835683;0.0334235[, [0.0334235;0.648386[, [0.
    ↪648386;1.44563[, [1.44563;12.1147]>>)
V5:Discretized(<(-22.1055;-0.813666[, [-0.813666;-0.355923[, [-0.355923;0.0329468[, [0.
    ↪0329468;0.534605[, [0.534605;11.9743]>>)
V6:Discretized(<(-7.5748;-0.789778[, [-0.789778;-0.370597[, [-0.370597;0.0353554[, [0.
    ↪0353554;0.711815[, [0.711815;10.0339]>>)
V7:Discretized(<(-43.5572;-0.691953[, [-0.691953;-0.264738[, [-0.264738;0.111993[, [0.
    ↪111993;0.57616[, [0.57616;12.2192]>>)
V8:Discretized(<(-41.0443;-0.248778[, [-0.248778;-0.0618973[, [-0.0618973;0.101159[, [0.
    ↪101159;0.417327[, [0.417327;20.0072]>>)
V9:Discretized(<(-13.4341;-0.258886[, [-0.258886;0.432783[, [0.432783;1.00315[, [1.00315;
    ↪1.60675[, [1.60675;10.3929]>>)
V10:Discretized(<(-24.5883;-0.887242[, [-0.887242;-0.486914[, [-0.486914;-0.17427[, [-0.
    ↪17427;0.281998[, [0.281998;12.2599]>>)
V11:Discretized(<(-2.59533;-0.21685[, [-0.21685;0.467606[, [0.467606;1.06928[, [1.06928;
    ↪1.89436[, [1.89436;12.0189]>>)
V12:Discretized(<(-18.6837;-2.60336[, [-2.60336;-1.98917[, [-1.98917;-1.01028[, [-1.
    ↪01028;0.297745[, [0.297745;3.77484]>>)
V13:Discretized(<(-3.38951;-0.277526[, [-0.277526;0.487335[, [0.487335;1.192[, [1.192;1.
    ↪87168[, [1.87168;4.46541]>>)
V14:Discretized(<(-19.2143;-0.198436[, [-0.198436;0.39438[, [0.39438;1.12921[, [1.12921;
    ↪1.5604[, [1.5604;5.74873]>>)
V15:Discretized(<(-4.49894;-0.898218[, [-0.898218;-0.252119[, [-0.252119;0.228109[, [0.
    ↪228109;0.673846[, [0.673846;2.53366]>>)
```

(continues on next page)

(continued from previous page)

```

V16:Discretized(<(-14.1299;-0.73753[, [-0.73753;-0.191439[, [-0.191439;0.226074[, [0.
↪ 226074;0.649708[, [0.649708;3.93088)>)
V17:Discretized(<(-25.1628;-0.37327[, [-0.37327;0.0631357[, [0.0631357;0.445363[, [0.
↪ 445363;0.906548[, [0.906548;7.89339)>)
V18:Discretized(<(-9.49875;-0.642528[, [-0.642528;-0.179343[, [-0.179343;0.16627[, [0.
↪ 16627;0.556347[, [0.556347;4.11556)>)
V19:Discretized(<(-4.93273;-0.673233[, [-0.673233;-0.228783[, [-0.228783;0.150301[, [0.
↪ 150301;0.636972[, [0.636972;5.22834)>)
V20:Discretized(<(-13.276;-0.183662[, [-0.183662;-0.0677703[, [-0.0677703;0.0444333[, [0.
↪ 0444333;0.232763[, [0.232763;11.059)>)
V21:Discretized(<(-22.7976;-0.298193[, [-0.298193;-0.179497[, [-0.179497;-0.0548622[, [-
↪ 0.0548622;0.105119[, [0.105119;27.2028)>)
V22:Discretized(<(-8.88702;-0.649014[, [-0.649014;-0.291808[, [-0.291808;0.00962799[, [0.
↪ 00962799;0.351258[, [0.351258;8.36199)>)
V23:Discretized(<(-19.2543;-0.215265[, [-0.215265;-0.0924607[, [-0.0924607;-1.53e-05[, [-
↪ 1.53e-05;0.122579[, [0.122579;13.8762)>)
V24:Discretized(<(-2.51238;-0.441547[, [-0.441547;-0.0137249[, [-0.0137249;0.248364[, [0.
↪ 248364;0.468669[, [0.468669;3.2002)>)
V25:Discretized(<(-4.78161;-0.238382[, [-0.238382;0.024469[, [0.024469;0.212094[, [0.
↪ 212094;0.411607[, [0.411607;5.52509)>)
V26:Discretized(<(-1.33856;-0.390763[, [-0.390763;-0.124995[, [-0.124995;0.128837[, [0.
↪ 128837;0.66481[, [0.66481;3.51735)>)
V27:Discretized(<(-7.9761;-0.0970824[, [-0.0970824;-0.0255692[, [-0.0255692;0.0347242[, [
↪ 0.0347242;0.216123[, [0.216123;4.17339)>)
V28:Discretized(<(-3.05408;-0.0430296[, [-0.0430296;0.00633497[, [0.00633497;0.0299365[,
↪ 0.0299365;0.111331[, [0.111331;4.86077)>)
Amount:Discretized(<(0;2.78[, [2.78;11.66[, [11.66;25.52[, [25.52;73.5[, [73.5;4002.88)>)

```

```
In [5]: gnb.showBN(BNTest.bn)
nbsphinx-code-borderwhite
```

```
In [6]: gnb.showBN(BNTest.MarkovBlanket)
nbsphinx-code-borderwhite
```

## Learn from array-likes

We use a method to transform the csv file in two array-likes in order to train from the same database.

```
In [7]: #we use now another method to learn the BN (MIIC)
BNTest= skbn.BNClassifier(learningMethod = 'MIIC', prior= 'Smoothing', priorWeight = 0.5,
↪ discretizationStrategy = 'quantile', usePR = True,
↪ significant_digit = 13)

xTrain, yTrain = BNTest.XYfromCSV(filename = 'res/creditCardTest.csv', target = 'Class'
↪ )
```

```
In [8]: BNTest.fit(xTrain, yTrain)
```

```
In [9]: gnb.showBN(BNTest.bn)
nbsphinx-code-borderwhite
```

```
In [10]: gnb.showBN(BNTest.MarkovBlanket)
nbsphinx-code-borderwhite
```

### Create a classifier from a Bayesian network

If we already have a Bayesian network with learned parameters, we can create a classifier that uses it. In this case we do not have to train the classifier on data since the Bayesian network is already trained.

```
In [11]: ClassfromBN = skbn.BNClassifier(significant_digit = 7)

In [12]: ClassfromBN.fromTrainedModel(bn = BNTest.bn, targetAttribute = 'Class',
    ↪targetModality = '1.0',
    ↪threshold = BNTest.threshold, variableList = xTrain.
    ↪columns.tolist())

In [13]: gnb.showBN(ClassfromBN.bn)
nbsphinx-code-borderwhite

In [14]: gnb.showBN(ClassfromBN.MarkovBlanket)
nbsphinx-code-borderwhite
```

Then, we work with functions from scikit-learn like score. We can also call it with a csv file or two array-likes.

```
In [15]: xTest, yTest = ClassfromBN.XYfromCSV(filename = 'res/creditCardTest.csv', target =
    ↪'Class')
```

### Prediction for classifier

#### Prediction with csv file

```
In [16]: scoreCSV1 = BNTest.score('res/creditCardTest.csv', y = yTest)
print("{0:.2f}% good predictions".format(100*scoreCSV1))
99.77% good predictions

In [17]: scoreCSV2 = ClassfromBN.score('res/creditCardTest.csv', y = yTest)
print("{0:.2f}% good predictions".format(100*scoreCSV2))
99.77% good predictions
```

#### Prediction with array-like

```
In [18]: scoreAR1 = BNTest.score(xTest, yTest)
print("{0:.2f}% good predictions".format(100*scoreAR1))
99.77% good predictions

In [19]: scoreAR2 = ClassfromBN.score(xTest, yTest)
print("{0:.2f}% good predictions".format(100*scoreAR2))
99.77% good predictions
```



## ROC and Precision-Recall curves with all methods

In addition (and of course), we can work with functions from pyAgrum (from `pyAgrum.lib.bn2roc`).

```
In [20]: BNTest.showROC_PR('res/creditCardTest.csv')
nbsphinx-code-borderwhite
```

```
In [ ]:
```

### 1.24.2 The BNDiscretizer Class

 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrum.org">https://agrum.org</a>)</p>	<p>(<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>)</p>
--	--	--

Most of the functionality of pyAgrum works only on discrete data. However data in the real world can be often continous. This class can be used to create dizcretized variables from continous data. Since this class was made for the purposes of the class BNClassifier, this class accepts data in the form of ndarrays. To transform data from a csv file to an ndarray we can use the function in BNClassifier `XYfromCSV`.

#### Creation of an instance and setting parameters

To create an instance of this class we need to specify the default parameters (the discretization method and the number of bins) for discretizing data. We create a discretizer which uses the EWD (Equal Width Discretization) method with 5 bins. The threshold is used for determining if a variable is already discretized. In this case, if a variable has more than 10 unique values we treat it as continous. we can use the `setDiscretizationParameters` method to set the discretization parameters for a specific variable

```
In [1]: import pyAgrum.skbn as skbn

discretizer=skbn.BNDiscretizer(defaultDiscretizationMethod='uniform',
↪ defaultNumberOfBins=5,discretizationThreshold=10)

discretizer.setDiscretizationParameters('var4','quantile',10)
discretizer.setDiscretizationParameters('var5','NoDiscretization',None)
```

#### Auditing data

To see how certain data will be treated by the discretizer we can use the audit method.

```
In [2]: import pandas
X = pandas.DataFrame.from_dict({
    'var1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3],
    'var2': ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n'],
    'var3': [1, 2, 5, 1, 2, 5, 1, 2, 5, 1, 2, 5, 1, 2],
    'var4': [1.11, 2.213, 3.33, 4.23, 5.42, 6.6, 7.5, 8.9, 9.19, 10.11, 11.12, 12.21, ↪
↪ 13.3, 14.5],
```

(continues on next page)

(continued from previous page)

```
'var5': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1]
})
```

```
print(X)
```

```
auditDict=discretizer.audit(X)
```

```
print()
```

```
print("*** audit ***")
```

```
for var in auditDict:
    print(f"- {var} : ")
    for k,v in auditDict[var].items():
        print(f"    + {k} : {v}")
```

	var1	var2	var3	var4	var5
0	1	a	1	1.110	1
1	2	b	2	2.213	2
2	3	c	5	3.330	3
3	4	d	1	4.230	4
4	5	e	2	5.420	5
5	6	f	5	6.600	6
6	7	g	1	7.500	7
7	8	h	2	8.900	8
8	9	i	5	9.190	9
9	10	j	1	10.110	10
10	11	k	2	11.120	11
11	1	l	5	12.210	12
12	2	m	1	13.300	13
13	3	n	2	14.500	1

```
** audit **
```

```
- var1 :
    + k : 5
    + methode : uniform
    + type : Continuous
- var2 :
    + methode : NoDiscretization
    + k : 14
    + type : Discrete
- var3 :
    + methode : NoDiscretization
    + k : 3
    + type : Discrete
- var4 :
    + k : 10
    + methode : quantile
    + type : Continuous
- var5 :
    + k : None
    + methode : NoDiscretization
    + type : Discrete
```

We can see that even though var2 has more unique values than var1 it is treated as a discrete variable. This is because the values of var2 are strings and therefore cannot be discretized.

## Creating variables from data

To create variables from data we can use the `createVariable` method for each column in our data matrix. This will use the parameters that we have already set to create discrete (or discretized) variables from our data.

```
In [3]: var1=discretizer.createVariable('var1',X['var1'])
var2=discretizer.createVariable('var2',X['var2'])
var3=discretizer.createVariable('var3',X['var3'])
var4=discretizer.createVariable('var4',X['var4'])
var5=discretizer.createVariable('var5',X['var5'])

print(var1)
print(var2)
print(var3)
print(var4)
print(var5)

var1:Discretized(<(1;3[, [3;5[, [5;7[, [7;9[, [9;11)>)
var2:Labelized({a|b|c|d|e|f|g|h|i|j|k|l|m|n})
var3:Integer({1|2|5})
var4:Discretized(<(1.11;2.5481[, [2.5481;3.87[, [3.87;5.301[, [5.301;6.78[, [6.78;8.2[, [8.
↪2;9.132[, [9.132;10.211[, [10.211;11.556[, [11.556;12.973[, [12.973;14.5)>)
var5:Range([1,13])
```

For supervised discretization algorithms (MDLP and CAIM) the list of class labels for each datapoint is also needed.

```
In [4]: y=[True,False,False,True,False,False,True,True,False,False,True,True,False,True]

discretizer.setDiscretizationParameters('var4','CAIM')
var4=discretizer.createVariable('var4',X['var4'],y)
print(var4)
discretizer.setDiscretizationParameters('var4','MDLP')
var4=discretizer.createVariable('var4',X['var4'],y)
print(var4)

var4:Discretized(<(1.11;10.615[, [10.615;14.5)>)
var4:Discretized(<(1.11;1.6615[, [1.6615;14.5)>)
```

The discretizer keeps track of the number of discretized variables created by it and the number of bins used to discretize them. To reset these two numbers to 0 we can use the `clear` method. We can also use it to clear the specific parameters we have set for each variable.

```
In [5]: print(f"numberOfContinuous : {discretizer.numberOfContinuous}")
print(f"totalNumberOfBins : {discretizer.totalNumberOfBins}")

discretizer.clear()
print("\n")

print(f"numberOfContinuous : {discretizer.numberOfContinuous}")
print(f"totalNumberOfBins : {discretizer.totalNumberOfBins}")

discretizer.audit(X)

numberOfContinuous : 4
totalNumberOfBins : 19

numberOfContinuous : 0
totalNumberOfBins : 0
```

```
Out[5]: {'var1': {'k': 5, 'methode': 'uniform', 'type': 'Continuous'},
        'var2': {'methode': 'NoDiscretization', 'k': 14, 'type': 'Discrete'},
        'var3': {'methode': 'NoDiscretization', 'k': 3, 'type': 'Discrete'},
        'var4': {'k': 10, 'methode': 'MDLP', 'type': 'Continuous'},
        'var5': {'k': None, 'methode': 'NoDiscretization', 'type': 'Discrete'}}
```

```
In [6]: discretizer.clear(True)
        discretizer.audit(X)
```

```
Out[6]: {'var1': {'k': 5, 'methode': 'uniform', 'type': 'Continuous'},
        'var2': {'methode': 'NoDiscretization', 'k': 14, 'type': 'Discrete'},
        'var3': {'methode': 'NoDiscretization', 'k': 3, 'type': 'Discrete'},
        'var4': {'k': 5, 'methode': 'uniform', 'type': 'Continuous'},
        'var5': {'k': 5, 'methode': 'uniform', 'type': 'Continuous'}}
```

### Using Discretizer with BNLearner

```
In [7]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

import pyAgrum.skbn as skbn
import pandas as pd

file_name = 'res/discretizable.csv'
data = pd.read_csv(file_name)

discretizer = skbn.BNDiscretizer(defaultDiscretizationMethod='uniform',
                                defaultNumberOfBins=5,
                                discretizationThreshold=25)
```

```
In [8]: template = gum.BayesNet()
for name in data:
    template.add(discretizer.createVariable(name, data[name]))
```

```
In [9]: learner = gum.BNLearner(file_name, template)
learner.useMIIC()
learner.useNMLCorrection()



bn = learner.learnBN()
gnb.show(bn, size="10!")
nbsphinx-code-borderwhite
```

```
In [ ]:
```

### 1.24.3 Comparing classifiers (including Bayesian networks) with scikit-learn

In this notebook, we use the skbn module to insert bayesian networks into some examples from the scikit-learn documentation (that we refer).



 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org">           (https://agrum.org)         </a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">           (https://agrum.gitlab.io/extra/agrum_at_binder.html)         </a>
--	--	---

```
In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
from pyAgrum.skbn import BNClassifier
```

### Binary classifiers

```
In [2]: # From https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_
↳ comparison.html)
# Code source: Gael Varoquaux
#             Andreas Muller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause
```

```
In [3]: import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patheffects as pe

from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

```
In [4]: # the data
X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]
datasets_name=['Moons ',
```

(continues on next page)

(continued from previous page)

```
'Circle',
'LinSep']
```

```
In [5]: def showComparison(names,classifiers,datasets,datasets_name):# the results
bnres=[None]*len(datasets_name)
h = .02 # step size in the mesh
fs=6

figure = plt.figure(figsize=(10, 4))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    print(datasets_name[ds_cnt]+' : ',end='')
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.4, random_state=42)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                        np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data",fontsize=fs)
        ax.set_ylabel(datasets_name[ds_cnt])

    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k',marker=".")
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
               edgecolors='k',marker=".")
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        print(".",end="",flush=True)
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
```

(continues on next page)

(continued from previous page)

```

        Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.7)

        # Plot the training points
        #ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
        #          edgecolors='k', alpha=0.2, marker='.')
        # Plot the testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                  edgecolors='k', marker='.')

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title(name, fontsize=fs)
            ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
                    size=12, horizontalalignment='right', color="white",
                    path_effects=[pe.withStroke(linewidth=2, foreground="black")])
            i += 1
        bnres[ds_cnt]=gum.BayesNet(classifiers[-1].bn)
        print()

    plt.tight_layout()
    plt.show()

    return bnres

```

```

In [6]: # the classifiers
names = ["Nearest Neighbors", "Linear SVM", "RBF SVM", "Gaussian Process",
        "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
        "Naive Bayes", "QDA", "BNClassifier"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
    BNClassifier(learningMethod='MIIC', prior='Smoothing', priorWeight=1,
    ↳ discretizationNbBins=5, discretizationStrategy="uniform", usePR=False)
]

bnres=showComparison(names,classifiers,datasets,datasets_name)

Moons : ...
Circle : ...
LinSep : ...

```

nbsphinx-code-borderwhite

The three BNs learned for each task:

```
In [7]: gnb.sideBySide(*bnres,captions=datasets_name)
```

```
<IPython.core.display.HTML object>
```

Note that, for LinSep, the BNClassifier has correctly learned that  $x_1$  and  $y$  are independent (no need of  $x_1$  to predict  $y$ ).  $x_1$  is not a relevant feature for this classification.

### A zoom of one of this BN classifiers

```
In [8]: h=0.2
ds=make_moons(noise=0.3, random_state=0)
X, y = ds
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test =train_test_split(X, y, test_size=.4, random_
↪state=42)
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
clf=BNClassifier(learningMethod='MIIC', prior='Smoothing', priorWeight=1,
↪discretizationNbBins=5,discretizationStrategy="uniform",usePR=False)

clf.fit(X_train,y_train)
score = clf.score(X_test, y_test)

Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
clf.bn

Z = Z.reshape(xx.shape)

ax = plt.subplot(1, 1,1)

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xticks(())
ax.set_yticks(())

ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,  alpha=0.6,edgecolors='k',marker=".")
ax.contourf(xx, yy, Z,  alpha=.7)

ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
        size=12, horizontalalignment='right',color="white",
        path_effects=[pe.withStroke(linewidth=2, foreground="black")]);
```

nbsphinx-code-borderwhite

## n-ary classifiers on IRIS dataset

```
In [9]: # From https://scikit-learn.org/stable/auto_examples/classification/plot_
↪classification_probability.html#sphx-glr-auto-examples-classification-plot-
↪classification-probability-py
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
# License: BSD 3 clause
```

```
In [10]: import matplotlib.pyplot as plt
import numpy as np

from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, 0:2] # we only take the first two features for visualization
y = iris.target

n_features = X.shape[1]

C = 10
kernel = 1.0 * RBF([1.0, 1.0]) # for GPC

# Create different classifiers.
classifiers = {
    'L1 logistic': LogisticRegression(C=C, penalty='l1',
                                     solver='saga',
                                     multi_class='multinomial',
                                     max_iter=10000),
    'L2 logistic (Multinomial)': LogisticRegression(C=C, penalty='l2',
                                                    solver='saga',
                                                    multi_class='multinomial',
                                                    max_iter=10000),
    'L2 logistic (OvR)': LogisticRegression(C=C, penalty='l2',
                                           solver='saga',
                                           multi_class='ovr',
                                           max_iter=10000),
    'Linear SVC': SVC(kernel='linear', C=C, probability=True,
                     random_state=0),
    'GPC': GaussianProcessClassifier(kernel),
    'BN' : BNClassifier(learningMethod='MIIC',
                       prior='Smoothing', priorWeight=1,
                       discretizationNbBins=6,
                       discretizationStrategy="kmeans",
                       discretizationThreshold=10)
}

n_classifiers = len(classifiers)

plt.figure(figsize=(3 * 2, n_classifiers * 2))
plt.subplots_adjust(bottom=.2, top=.95)

xx = np.linspace(3, 9, 100)
```

(continues on next page)

(continued from previous page)

```

yy = np.linspace(1, 5, 100).T
xx, yy = np.meshgrid(xx, yy)
Xfull = np.c_[xx.ravel(), yy.ravel()]

for index, (name, classifier) in enumerate(classifiers.items()):
    classifier.fit(X, y)

    y_pred = classifier.predict(X)
    accuracy = accuracy_score(y, y_pred)
    print("Accuracy (train) for %s: %0.1f%%" % (name, accuracy * 100))

    # View probabilities:
    probas = classifier.predict_proba(Xfull)
    n_classes = np.unique(y_pred).size
    for k in range(n_classes):
        plt.subplot(n_classifiers, n_classes, index * n_classes + k + 1)
        plt.title("Class %d" % k)
        if k == 0:
            plt.ylabel(name)
        imshow_handle = plt.imshow(probas[:, k].reshape((100, 100)),
                                   extent=(3, 9, 1, 5), origin='lower')

        plt.xticks(())
        plt.yticks(())
        idx = (y_pred == k)
        if idx.any():
            plt.scatter(X[idx, 0], X[idx, 1], marker='o', c='w', edgecolor='k')

    ax = plt.axes([0.15, 0.04, 0.7, 0.05])
    plt.title("Probability")
    plt.colorbar(imshow_handle, cax=ax, orientation='horizontal')

plt.show()

```

```

Accuracy (train) for L1 logistic: 82.7%
Accuracy (train) for L2 logistic (Multinomial): 82.7%
Accuracy (train) for L2 logistic (OvR): 79.3%
Accuracy (train) for Linear SVC: 82.0%
Accuracy (train) for GPC: 82.7%
Accuracy (train) for BN: 83.3%

```

nbsphinx-code-borderwhite

So the BNClassifier gives the ‘best’ accuracy (even if discretized). Moreover, once again, it propose a structural representation of the classification mechanism.

```
In [11]: classifiers['BN'].bn
```

```
Out[11]: (pyAgrum.BayesNet<double>@00000201020A7680) BN{nodes: 3, arcs: 2, domainSize: 108,
↪dim: 120}
```

## Recognizing hand-written digits with Bayesian Networks

```
In [12]: #From https://scikit-learn.org/stable/auto_examples/classification/plot_digits_
↳classification.html#sphx-glr-auto-examples-classification-plot-digits-
↳classification-py
# Author: Gael Varoquaux <gael dot varoquaux at normalesup dot org>
# License: BSD 3 clause
```

```
In [13]: # Standard scientific Python imports
import matplotlib.pyplot as plt

# Import datasets, classifiers and performance metrics
from sklearn import datasets, metrics
from sklearn.model_selection import train_test_split

digits = datasets.load_digits()

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, label in zip(axes, digits.images, digits.target):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)
```

nbsphinx-code-borderwhite

```
In [14]: # flatten the images
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1)).astype(int)

# Create a classifier: a support vector classifier
#clf = svm.SVC(gamma=0.001)
clf = BNClassifier(learningMethod='MIIC', prior='Smoothing', priorWeight=1,
                  discretizationNbBins=3,discretizationStrategy="kmeans",
↳discretizationThreshold=10)

# Split data into 50% train and 50% test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# Learn the digits on the train subset
clf.fit(X_train, y_train)

# Predict the value of the digit on the test subset
predicted = clf.predict(X_test)

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, prediction in zip(axes, X_test, predicted):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title(f'Prediction: {prediction}')

cm = metrics.confusion_matrix(y_test, predicted)
disp = metrics.ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()

plt.show()
```

(continues on next page)

(continued from previous page)

```
print(f"Classification report for classifier {clf}:\n"
      f"{metrics.classification_report(y_test, predicted)}\n")
```

```
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
```

```
Classification report for classifier BNClassifier(prior='Smoothing',
↳ discretizationNbBins=3,
          discretizationStrategy='kmeans', discretizationThreshold=10,
          learningMethod='MIIC'):
```

	precision	recall	f1-score	support
0	0.98	0.95	0.97	88
1	0.86	0.80	0.83	91
2	0.91	0.85	0.88	86
3	0.84	0.80	0.82	91
4	0.99	0.91	0.95	92
5	0.79	0.82	0.81	91
6	0.95	0.95	0.95	91
7	0.89	0.90	0.89	89
8	0.79	0.76	0.77	88
9	0.72	0.90	0.80	92
accuracy			0.87	899
macro avg	0.87	0.87	0.87	899
weighted avg	0.87	0.87	0.87	899

### Focus on the pixels needed for the classification

As always, using BNClassifier make us learn a bit more about the structure of the problem.

```
In [15]: gnb.show(clf.bn,size="13!")
```

```
nbsphinx-code-borderwhite
```

Then, once again, the Markov Blanket gives us the relevant features (here the pixels)

```
In [16]: print("Markov blanket of the classifier:")
gnb.show(clf.MarkovBlanket,size="14!")
print(f"Number of pixels used for classification : {clf.MarkovBlanket.size()-1}/64")
```

```
Markov blanket of the classifier:
```

```
nbsphinx-code-borderwhite
```

```
Number of pixels used for classification : 33/64
```



It appears that many pixels are not relevant for this classification.

```
In [17]: #Visualization of the pixels of the Markov Blanket
```

```
fig, ax = plt.subplots()
ax.set_axis_off()
relevant_pixels = set([int(x[1:]) for x in clf.MarkovBlanket.names() if x!='y'])
ax.imshow(np.array([1 if i in relevant_pixels else 0 for i in range(64)]).reshape(8,
↳ 8),
          cmap=plt.cm.gray_r)
plt.show()
```



The purpose of this notebook is to show the possible integration of the pyAgrum's classifier in the scikit-learn's ecosystem. Thus, it is possible to use the tools provided by scikit-learn for crossfolding for pyAgrum's Bayesian network.

 ( <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a> )	 ( <a href="https://agrum.org">https://agrum.org</a> )	( <a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a> )
--	--	---

```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
from pyAgrum.skbn import BNClassifier
```

```
from sklearn.model_selection import cross_validate
from sklearn import datasets
# get iris data
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
model = BNClassifier(learningMethod='MIIC', prior='Smoothing', priorWeight=1,
                    discretizationNbBins=3, discretizationStrategy="kmeans",
                    discretizationThreshold=10)
```

```
cv = cross_validate(model, X, y, cv=30)
print(f"scores with cross-folding : {cv['test_score']}")
print()

print(f"mean score : {cv['test_score'].mean()}")
```

scores with cross-folding : [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0.8 1. 1. 1. 1.  
↪0.8 1. 1. 0.8  
1. 1. 1. 0.8 1. 1. 1. 1. 1. 1. 1. ]

mean score : 0.9733333333333333

```
cv = cross_validate(model, X, y, cv=50)
print(f"scores with cross-folding : {cv['test_score']}")
print()

print(f"mean score : {cv['test_score'].mean()}")
```

scores with cross-folding : [1. 1. 1. 1. 1. 1.  
1. 1. 1. 1. 1. 1.]

(continues on next page)

(continued from previous page)

```

1.      1.      1.      1.      1.      1.
1.      1.      0.66666667 1.      1.      1.
1.      1.      1.      0.66666667 1.      1.
1.      1.      1.      0.66666667 1.      1.
1.      1.      1.      1.      1.      1.
1.      1.      1.      1.      1.      1.
1.      1.      ]

```

```
mean score : 0.98
```

In [ ]:

### 1.24.5 From a Bayesian network to a Classifier

This notebook shows how to build a classifier from an Bayesian network (and not from a database).



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: import pyAgrum as gum
import pyAgrum.skbn as skbn
import pyAgrum.lib.notebook as gnb
```

```
In [2]: bn=gum.loadBN("res/alarm.dsl")
gnb.showBN(bn,size="10")
print(bn.variable("HR"))
nbsphinx-code-borderwhite
HR:Labelized({LOW|NORMAL|HIGH})
```

Let's say that you would like to use this Bayesian network to learn a classifier for the class HR (3 classes)

```
In [3]: #generating the base of 10 values for testing purpose
print(f"LL(alarm-10)={gum.generateSample(bn,100,'out/alarm-10.csv')}")
LL(alarm-10)=-1523.1404258054915
```

```
In [4]: bnc=skbn.BNClassifier()
bnc.fromTrainedModel(bn,targetAttribute="HR")
print(f"Binary classifier : {bnc.isBinaryClassifier}")
gnb.showBN(bnc.MarkovBlanket)

xTrain, yTrain = bnc.XYfromCSV(filename ='out/alarm-10.csv' )
print(f"predicted : {list(bnc.predict(xTrain))}")
print(f"in base   : {yTrain.to_list()}")

Binary classifier : False
nbsphinx-code-borderwhite
```

```

predicted : ['NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH',
→ 'HIGH', 'NORMAL', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'NORMAL',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH']
in base : ['NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH',
→ 'HIGH', 'NORMAL', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'LOW',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH']

```

```
In [5]: print(list(bnc.predict(X='out/alarm-10.csv')))
```

```

['NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH',
→ 'NORMAL', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH',
→ 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'LOW', 'HIGH', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH',
→ 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'NORMAL', 'HIGH', 'HIGH', 'HIGH', 'HIGH']

```

```
In [6]: scoreCSV1 = bnc.score('out/alarm-10.csv', y = yTrain)
print("{0:.2f}% good predictions".format(100*scoreCSV1))
```

```
99.00% good predictions
```

## From a Bayesian network to a Binary classifier

By targetting a specific label, one can create a binary classifier to predict this very target.

```
In [7]: bnc=skbn.BNClassifier()
bnc.fromTrainedModel(bn,targetAttribute="HR",targetModality="LOW")
print(f"Binary classifier : {bnc.isBinaryClassifier}")
gnb.showBN(bnc.MarkovBlanket)

xTrain, yTrain = bnc.XYfromCSV(filename='out/alarm-10.csv')
print(f"predicted : {list(bnc.predict(xTrain))}")
print(f"in base : {yTrain.to_list()}")
```

```
Binary classifier : True
```

```
nbsphinx-code-borderwhite
```

```

predicted : [False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, True, False, False, False]
in base : [False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳True, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, True, False, False, False]

```

```
In [8]: print(list(bnc.predict(X='out/alarm-10.csv')))
```

```

[False, False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳False, False, False, False, False, False, False, False, False, False, False,
↳True, False, False, False]

```

```
In [9]: scoreCSV1 = bnc.score('out/alarm-10.csv', y = yTrain)
print("{0:.2f}% good predictions".format(100*scoreCSV1))
```

```
99.00% good predictions
```

```
In [10]: print(f"LL(alarm-1000)={gum.generateSample(bn,1000,'out/alarm-1000.csv',with_
↳labels=True)}")
bnc.showROC_PR('out/alarm-1000.csv')
```

```
LL(alarm-1000)=-14788.400035254197
```

```
nbsphinx-code-borderwhite
```



```
In [ ]:
```

## 1.25 Causal Bayesian Networks

### 1.25.1 Smoking, Cancer and causality

This notebook follows the famous example from *Causality* (Pearl, 2009).

A correlation has been observed between Smoking and Cancer, represented by this Bayesian network :

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org">           (https://agrum.org)         </a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">           (https://agrum.gitlab.io/extra/agrum_at_binder.html)         </a>
--	--	---

```
In [1]: from IPython.display import display, Math, Latex, HTML

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.causal as csl
import pyAgrum.causal.notebook as cslnb

obs1 = gum.fastBN("Smoking->Cancer")

obs1.cpt("Smoking")[:] = [0.6, 0.4]
obs1.cpt("Cancer")[{ "Smoking": 0 }] = [0.9, 0.1]
obs1.cpt("Cancer")[{ "Smoking": 1 }] = [0.7, 0.3]

gnb.flow.row(obs1, obs1.cpt("Smoking") * obs1.cpt("Cancer"), obs1.cpt("Smoking"), obs1.cpt(
    ↪ "Cancer"),
             captions=["the BN", "the joint distribution", "the marginal for $smoking$
    ↪ ", "the CPT for $cancer$"])

<IPython.core.display.HTML object>
```

### Direct causality between Smoking and Cancer

The very strong observed correlation between smoking and lung cancer suggests a causal relationship as the Surgeon General asserts in 1964, then, the proposed model is as follows :

```
In [2]: # the Bayesian network is causal
modele1 = csl.CausalModel(obs1)

cslnb.showCausalImpact(modele1, "Cancer", "Smoking", values={"Smoking": 1})

<IPython.core.display.HTML object>
```

### Latent confounder between Smoking and Cancer

This model is highly contested by the tobacco industry which answers by proposing a different model in which Smoking and Cancer are simultaneously provoked by a common factor, the Genotype (or other latent variable) :

```
In [3]: # a latent variable exists between Smoking and Cancer in the causal model
modele2 = csl.CausalModel(obs1, [("Genotype", ["Smoking", "Cancer"])]

cslnb.showCausalImpact(modele2, "Cancer", "Smoking", values={"Smoking":1})

<IPython.core.display.HTML object>
```

```
In [4]: # just check P(Cancer) in the bn `obs1`
(obs1.cpt("Smoking")*obs1.cpt("Cancer")).margSumIn(["Cancer"])
```

```
Out[4]: (pyAgrum.Potential<double>@0000029C313077C0)
      Cancer      |
0      | 1      |
-----|-----|
0.8200 | 0.1800 |
```

### Confounder and direct causality

In a diplomatic effort, both parts agree that there must be some truth in both models :

```
In [5]: # a latent variable exists between Smoking and Cancer but the direct causal relation
        ↪ exists also
modele3 = csl.CausalModel(obs1, [("Genotype", ["Smoking", "Cancer"])], True)

cslnb.showCausalImpact(modele3, "Cancer", "Smoking", values={"Smoking":1})

<IPython.core.display.HTML object>
```

Smoking's causal effect on Cancer becomes uncomputable in such a model because we can't distinguish both causes' impact from the observations.

### A intermediary observed variable

We introduce an auxiliary factor between Smoking and Cancer, tobacco causes cancer because of the tar deposits in the lungs.

```
In [6]: obs2 = gum.fastBN("Smoking->Tar->Cancer;Smoking->Cancer")

obs2.cpt("Smoking")[:] = [0.6, 0.4]
obs2.cpt("Tar") [{"Smoking": 0}] = [0.9, 0.1]
obs2.cpt("Tar") [{"Smoking": 1}] = [0.7, 0.3]
obs2.cpt("Cancer") [{"Tar": 0, "Smoking": 0}] = [0.9, 0.1]
obs2.cpt("Cancer") [{"Tar": 1, "Smoking": 0}] = [0.8, 0.2]
obs2.cpt("Cancer") [{"Tar": 0, "Smoking": 1}] = [0.7, 0.3]
obs2.cpt("Cancer") [{"Tar": 1, "Smoking": 1}] = [0.6, 0.4]

gnb.flow.row(obs2, obs2.cpt("Smoking"), obs2.cpt("Tar"), obs2.cpt("Cancer"),
             captions=["", "$P(Smoking)$", "$P(Tar|Smoking)$", "$P(Cancer|Tar, Smoking)$",
             ↪ ""])
```

```
<IPython.core.display.HTML object>
```

```
In [7]: modele4 = csl.CausalModel(obs2, [("Genotype", ["Smoking", "Cancer"])]
cslnb.showCausalModel(modele4)
nbsphinx-code-borderwhite
```

```
In [8]: cslnb.showCausalImpact(modele4, "Cancer", "Smoking", values={"Smoking":1})

<IPython.core.display.HTML object>
```

In this model, we are, again, able to calculate the causal impact of Smoking on Cancer thanks to the verification of the Frontdoor criterion by the Tar relatively to the couple (Smoking, Cancer)

```
In [9]: # just check P(Cancer|do(smoking)) in the bn `obs2`
((obs2.cpt("Cancer")*obs2.cpt("Smoking")).margSumOut(["Smoking"])*obs2.cpt("Tar")).
↪margSumOut(['Tar']).putFirst("Cancer")
```

```
Out[9]: (pyAgrum.Potential<double>@00000029C31307400)
```

	Cancer	
Smokin	0	1
0	0.8100	0.1900
1	0.7900	0.2100

### Other causal impacts for this last model

```
In [10]: cslnb.showCausalImpact(modele4, "Smoking", doing="Cancer", knowing={"Tar"}, values={
↪ "Cancer":1, "Tar":1})
```

```
<IPython.core.display.HTML object>
```

```
In [11]: cslnb.showCausalImpact(modele4, "Smoking", doing="Cancer", values={"Cancer":1})
```

```
<IPython.core.display.HTML object>
```

```
In [12]: cslnb.showCausalImpact(modele4, "Smoking", doing={"Cancer", "Tar"}, values={"Cancer":1,
↪ "Tar":1})
```



```
<IPython.core.display.HTML object>
```

```
In [13]: cslnb.showCausalImpact(modele4, "Tar", doing={"Cancer", "Smoking"}, values={"Cancer":1,
↪ "Smoking":1})
```

```
<IPython.core.display.HTML object>
```

## 1.25.2 Simpson's Paradox

This notebook follows the famous example from *Causality* (Pearl, 2009).

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org"> (https://agrum.org) </a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html) </a>
--	--	---

In [1]: `from IPython.display import display, Math, Latex`

```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.causal as csl
import pyAgrum.causal.notebook as cslnb
```

In a statistical study about a drug, we try to evaluate the latter's efficiency among a population of men and women. Let's note: - *Drug* : drug taking - *Patient* : cured patient - *Gender* : patient's gender

The model from the observed date is as follow :

```
In [2]: m1 = gum.fastBN("Gender{F|M}->Drug{Without|With}->Patient{Sick|Healed}<-Gender")

m1.cpt("Gender")[:] = [0.5, 0.5]
m1.cpt("Drug")[:] = [[0.25, 0.75], #Gender=F
                    [0.75, 0.25]] #Gender=M

m1.cpt("Patient") [{'Drug': 'Without', 'Gender': 'F'}] = [0.2, 0.8] #No Drug, Male -> healed
↳ in 0.8 of cases
m1.cpt("Patient") [{'Drug': 'Without', 'Gender': 'M'}] = [0.6, 0.4] #No Drug, Female ->
↳ healed in 0.4 of cases
m1.cpt("Patient") [{'Drug': 'With', 'Gender': 'F'}] = [0.3, 0.7] #Drug, Male -> healed 0.7
↳ of cases
m1.cpt("Patient") [{'Drug': 'With', 'Gender': 'M'}] = [0.8, 0.2] #Drug, Female -> healed in
↳ 0.2 of cases
gnb.flow.row(m1, m1.cpt("Gender"), m1.cpt("Drug"), m1.cpt("Patient"))
```

<IPython.core.display.HTML object>

```
In [3]: def getCuredObservedProba(m1, evs):
    evs0 = dict(evs)
    evs1 = dict(evs)
    evs0["Drug"] = 'Without'
    evs1["Drug"] = 'With'

    return gum.Potential().add(m1.variableFromName("Drug")).fillWith([
        gum.getPosterior(m1, target="Patient", evs=evs0)[1],
        gum.getPosterior(m1, target="Patient", evs=evs1)[1]
    ])

gnb.sideBySide(getCuredObservedProba(m1, {}),
```

(continues on next page)



(continued from previous page)

```

getCuredObservedProba(m1,{ 'Gender': 'F' }),
getCuredObservedProba(m1,{ 'Gender': 'M' }),
captions=["$P(Patient = Healed \mid Drug )$<br/>Taking $Drug$ is_
↪observed as efficient to cure",
          "$P(Patient = Healed \mid Gender=F,Drug)$<br/>except if the
↪$gender$ of the patient is female",
          "$P(Patient = Healed \mid Gender=M,Drug)$<br/>... or male."])
<IPython.core.display.HTML object>

```

Those results form a paradox called Simpson paradox :

$$P(C \mid \neg D) = 0.5 < P(C \mid D) = 0.575$$

$$P(C \mid \neg D, G = \text{Male}) = 0.8 > P(C \mid D, G = \text{Male}) = 0.7$$

$$P(C \mid \neg D, G = \text{Female}) = 0.4 > P(C \mid D, G = \text{Female}) = 0.2$$

Actually, giving a drug is not an observation in our model but rather an intervention. What if we use intervention instead of observation ?

### How to compute causal impacts on the patient's health ?

We propose this causal model.

```

In [4]: d1 = csl.CausalModel(m1)
        cslnb.showCausalModel(d1)
        nbsphinx-code-borderwhite

```

#### Computing $P(\text{Patient} = \text{Healed} \mid \hookrightarrow \text{Drug} = \text{Without})$

```

In [5]: cslnb.showCausalImpact(d1, "Patient", doing="Drug", values={"Drug" : "Without"})
<IPython.core.display.HTML object>

```

We have,  $P(\text{Patient} = \text{Healed} \mid \hookrightarrow \text{Drug} = \text{without}) = 0.6$

#### Computing $P(\text{Patient} = \text{Healed} \mid \hookrightarrow \text{Drug} = \text{With})$

```

In [6]: d1 = csl.CausalModel(m1)
        cslnb.showCausalImpact(d1, "Patient", "Drug", values={"Drug" : "With"})
<IPython.core.display.HTML object>

```

And then :  $P(\text{Patient} = \text{Healed} \mid \text{' : nbsphinx-math : hookrightarrow' Drug} = \text{With}) = 0.45$

Therefore :  $P(\text{Patient} = \text{Healed} : \text{nbsphinx-math:mid' : nbsphinx-math:hookrightarrow Drug} = \text{Without}) = 0.6 > P(\text{Patient} = \text{Healed} : \text{nbsphinx-math:mid' : nbsphinx-math:hookrightarrow Drug} = \text{With}) = 0.45$

Which means that taking this drug would not enhance the patient's healing process, and it is better not to prescribe this drug for treatment.

### Simpson paradox solved by interventions

So to summarize, the paradox appears when wrongly dealing with observations on *Drug* :

```
In [7]: gnb.sideBySide(getCuredObservedProba(m1,{}),
                    getCuredObservedProba(m1,{'Gender':'F'}),
                    getCuredObservedProba(m1,{'Gender':'M'}),
                    captions=["$P(Patient = Healed \mid Drug )$<br/>Taking $Drug$ is_
↪observed as efficient to cure",
                            "$P(Patient = Healed \mid Gender=F,Drug)$<br/>except if the
↪$gender$ of the patient is female",
                            "$P(Patient = Healed \mid Gender=M,Drug)$<br/>... or male."])

<IPython.core.display.HTML object>
```

... and disappears when dealing with intervention on *Drug* :



```
In [8]: gnb.sideBySide(csl.causalImpact(d1,on="Patient",doing="Drug",values={"Patient":"Healed"
↪})[1],
                    csl.causalImpact(d1,on="Patient",doing="Drug",knowing={"Gender"},
↪values={"Patient":"Healed","Gender":"F"})[1],
                    csl.causalImpact(d1,on="Patient",doing="Drug",knowing={"Gender"},
↪values={"Patient":"Healed","Gender":"M"})[1],
                    captions=["$P(Patient = 1 \mid \hookrightarrow Drug )$<br/>Effectively
↪$Drug$ taking is not efficient to cure",
                            "$P(Patient = 1 \mid \hookrightarrow Drug, gender=F )$<br/>,
↪the $gender$ of the patient being female",
                            "$P(Patient = 1 \mid \hookrightarrow Drug, gender=M )$<br/>,
↪... or male."])

<IPython.core.display.HTML object>
```

In [ ]:

### 1.25.3 Multinomial Simpson Paradox

this notebook shows a model for a multinomial Simpson paradox.

 ( <a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a> )	 ( <a href="https://agrum.org">https://agrum.org</a> )	 ( <a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a> )
--	--	---

```
In [1]: import matplotlib.pyplot as plt
import random

import pandas as pd
import numpy as np

import pyAgrum as gum
```

(continues on next page)

(continued from previous page)

```
import pyAgrum.lib.notebook as gnb

import pyAgrum.causal as csl
import pyAgrum.causal.notebook as cslnb
```

## Building the models

```
In [2]: # building a model including a Simpson's paradox
def fillWithUniform(p,fmin=None,fmax=None):
    if fmin is None:
        vmin=0
    if fmax is None:
        vmax=p.variable(0).domainSize()-1

    mi=int(p.variable(0).numerical(0))
    ma=int(p.variable(0).numerical(p.variable(0).domainSize()-1))

    p.fillWith(0)

    I=gum.Instantiation(p)

    I.setFirst()
    while not I.end():
        vars={p.variable(i).name():p.variable(i).numerical(I.val(i)) for i in range(1,p.
        ↪nbrDim())}
        if fmin is not None:
            vmin=int(eval(fmin,None,vars))
        if fmax is not None:
            vmax=int(eval(fmax,None,vars))
        if vmin<mi:
            vmin=mi
        if vmin>ma:
            vmin=ma
        if vmax<mi:
            vmax=mi
        if vmax>ma:
            vmax=ma

        for pos in range(vmin,vmax+1):
            I.chgVal(0,pos)
            p.set(I,1)
            I.incNotVar(p.variable(0))
            p.normalizeAsCPT()

    size=70
    sizeZ=5
    bn=gum.fastBN(f"A[0,{size-1}]->B[0,{size-1}]<-C[0,{sizeZ-1}]->A")

    bn.cpt("C").fillWith(1).normalize()
    fillWithUniform(bn.cpt("A"),fmin="C*12",fmax="C*12+30")
    bn.cpt("B").fillWithFunction("5+C*4-int(A/8)",[0.05,0.2,0.5,0.2,0.05]);
```

```
In [3]: # generating a CSV, taking this model as the causal one.
gum.generateSample(bn,400,"out/sample.csv")
```

(continues on next page)

(continued from previous page)

```
df=pd.read_csv("out/sample.csv")
df.plot.scatter(x='A', y='B', c='C',colormap="tab20");
nbsphinx-code-borderwhite
```

```
In [4]: cm=cs1.CausalModel(bn)
_,p,_=cs1.causalImpact(cm,on="B",doing="A")
```

```
In [5]: # building an Markov-equivalent model, generating a CSV, taking this model as the
        ↪ causal one.
bn2=gum.BayesNet(bn)
bn2.reverseArc("C", "A")

gum.generateSample(bn2,400,"out/sample2.csv")
df2=pd.read_csv("out/sample2.csv")

cm2=cs1.CausalModel(bn2)
_,p2,_=cs1.causalImpact(cm2,on="B",doing="A")
```

**The observationnal model and its paradoxal structure (exactly the same with the second Markov-equivalent model)**

```
In [6]: gnb.flow.row(gnb.getBN(bn),
                    df.plot.scatter(x='A', y='B'),
                    df.plot.scatter(x='A', y='B', c='C',colormap="tab20"),
                    captions=["the observationnal model","the trend is increasing","the
        ↪ trend is decreasing for any value for C !"])
gnb.flow.row(gnb.getBN(bn2),
            df2.plot.scatter(x='A', y='B'),
            df2.plot.scatter(x='A', y='B', c='C',colormap="tab20"),
            captions=["the Markov-equivalent model","the trend is increasing","the
        ↪ trend is decreasing for any value for C !"])

<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
```

**The paradox is revealed in the trend of the inferred means : the means are increasing with the value of  $A$  except for any value of  $C$  ...**

```
In [7]: for v in [10,20,30]:
        gnb.flow.add_html(gnb.getPosterior(bn,target="B",evs={"A":v}),f"$P(B|A={v})$")
        gnb.flow.new_line()
        for v in [10,20,30]:
            gnb.flow.add_html(gnb.getPosterior(bn,target="B",evs={"A":v,"C":0}),f"P(B | $A={v},
        ↪ C=0)$")
            gnb.flow.new_line()
            for v in [10,20,30]:
                gnb.flow.add_html(gnb.getPosterior(bn,target="B",evs={"A":v,"C":2}),f"P(B | $A={v},
        ↪ C=2)$")
                gnb.flow.new_line()
                for v in [10,20,30]:
                    gnb.flow.add_html(gnb.getPosterior(bn,target="B",evs={"A":v,"C":4}),f"P(B | $A={v},
        ↪ C=4)$")
                    gnb.flow.new_line()
        gnb.flow.display()
```

```
<IPython.core.display.HTML object>
```

Now that the paradoxal structure is understood and the paradox is revealed, will we choose to observe  $C$  (or not) before deciding to increase or decrease  $A$  (with the goal to maximize  $B$ ) ?

Of course, it depends on the causal structure of the problem !

```
In [8]: gnb.flow.add_html(cslnb.getCausalModel(cm), "the first causal model")
        for v in [10, 20, 30]:
            gnb.flow.add_html(gnb.getProba(p.extract({'A': v})), f"Doing $A={v}$")
        gnb.flow.display()
```

```
<IPython.core.display.HTML object>
```

**If  $C$  is cause for  $A$ , observing  $C$  really gives a new information about  $B$ .**

```
In [9]: gnb.flow.add_html(cslnb.getCausalModel(cm2),"the second causal model")
        for v in [10,20,30]:
            gnb.flow.add_html(gnb.getProba(p2.extract({'A':v})),f"Doing $A={v}$")
        gnb.flow.display()
```

```
<IPython.core.display.HTML object>
```

**if  $A$  is cause for  $C$ , observing  $C$  may lead to misinterpretations about the causal role of  $A$ .**

```
In [ ]:
```

### 1.25.4 Some examples of do-calculus



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: from IPython.display import display, Math, Latex, HTML

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.causal as csl
import pyAgrum.causal.notebook as cslnb
```

## S. Tikka and J. Karvanen, 2016 [CRAN]

```
In [2]: bn = gum.fastBN("w->x->z->y;w->z")

bn.cpt("w")[:] = [0.7,0.3]

bn.cpt("x")[:] = [[0.4,0.6],[0.3,0.7]]

bn.cpt("z")[{ 'w':0, 'x':0}]=[0.2,0.8]
bn.cpt("z")[{ 'w':0, 'x':1}]=[0.1,0.9]
bn.cpt("z")[{ 'w':1, 'x':0}]=[0.9,0.1]
bn.cpt("z")[{ 'w':1, 'x':1}]=[0.5,0.5]

bn.cpt("y")[:] = [[0.1,0.9],[0.8,0.2]]

d = csl.CausalModel(bn, [("lat1", ["x","y"])])
#csl.causalImpact(d, "y", {"x":0})
cslnb.showCausalImpact(d, "y", "x", values={"x":0})
cslnb.showCausalImpact(d, "y", "x", values={"x":1})
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Since we have the formula, let us compute by hand this intervention :

```
In [3]: (((bn.cpt("x") * bn.cpt("y")).margSumOut(["x"]) * bn.cpt("w") * bn.cpt("z")).
↪margSumOut(["z", "w"])).putFirst("y")
```

```
Out[3]: (pyAgrum.Potential<double>@00000019F24E70C80)
```

	y		
x	0	1	
0	0.5130	0.4870	
1	0.6460	0.3540	

```
In [4]: bn = gum.fastBN("Z1->X->Z2->Y")
d=csl.CausalModel(bn, [("L1", ["Z1","X"]),
                        ("L2", ["Z1","Z2"]),
                        ("L3", ["Z1","Y"]),
                        ("L4", ["Y","X"])],
                  True)
cslnb.showCausalImpact(d, "Y", "X", values={"X":1})

<IPython.core.display.HTML object>
```

## Front door

```
In [5]: modele4 = gum.BayesNet()
modele4.add(gum.LabelizedVariable("Smoking"))
modele4.add(gum.LabelizedVariable("Cancer"))
modele4.add(gum.LabelizedVariable("Tar"))

modele4.addArc(0,2)
modele4.addArc(2,1)
modele4.addArc(0,1)

#Smoking
modele4.cpt(0)[:]=[0.5, 0.5]

#Tar
modele4.cpt(2)[{"Smoking":0}] = [0.4, 0.6]
modele4.cpt(2)[{"Smoking":1}] = [0.3, 0.6]

#Cancer
modele4.cpt(1)[{'Smoking':0,'Tar':0}]=[0.1,0.9] #No Drug, Male -> healed in 0.8 of
↳ cases
modele4.cpt(1)[{'Smoking':0,'Tar':1}]=[0.15,0.85] #No Drug, Female -> healed in 0.4
↳ of cases
modele4.cpt(1)[{'Smoking':1,'Tar':0}]=[0.2,0.8] #Drug, Male -> healed 0.7 of cases
modele4.cpt(1)[{'Smoking':1,'Tar':1}]=[0.25,0.75]

d4 = csl.CausalModel(modele4, [("Genotype", ["Smoking","Cancer"])],False)
cslnb.showCausalModel(d4)
nbsphinx-code-borderwhite
```

```
In [6]: try:
        a = csl.doCalculusWithObservation (d4,"Cancer", {"Smoking"})
except csl.HedgeException as h:
    print (h.message)
```

```
In [7]: display(Math(a.toLatex()))
```

$$P(Cancer \mid \hookrightarrow Smoking) = \sum_{Tar} P(Tar \mid Smoking) \cdot \left( \sum_{Smoking'} P(Smoking') \cdot P(Cancer \mid Smoking', Tar) \right)$$

nbsphinx-code-borderwhite

```
In [8]: try:
        adjj = a.eval()
except csl.UnidentifiableException as u:
    print (u.message)

print (adjj)
```

	Cancer	
Smokin	0	1
0	0.1774	0.8226
1	0.1626	0.7374

```
In [9]: formula, adj, exp = csl.causalImpact(d4, "Cancer", "Smoking", values={"Smoking":0})
```

```
In [10]: display(Math(formula.toLatex()))
adj
```

$$P(Cancer \mid \hookrightarrow Smoking) = \sum_{Tar} P(Tar \mid Smoking) \cdot \left( \sum_{Smoking'} P(Cancer \mid Smoking', Tar) \cdot P(Smoking') \right)$$

nbsphinx-code-borderwhite

```
Out[10]: (pyAgrum.Potential<double>@00000019F24E6FF40)
```

Cancer		
0	1	
-----	-----	
0.1774	0.8226	

## Last example from R

```
In [11]: m = gum.fastBN("z2->x->z1->y;z2->z1;z2->z3->y")
```

```
m.cpt("z2") [:] = [0.5, 0.5]
m.cpt("x") [:] = [[0.4,0.6], #z2=0
                  [0.4,0.6]] #z2=1
m.cpt("z3") [:] = [[0.3,0.7], #z2=0
                  [0.3,0.7]] #z2=1
m.cpt("z1") [{"z2":0, "x":0}] = [0.2, 0.8]
m.cpt("z1") [{"z2":0, "x":1}] = [0.25, 0.75]
m.cpt("z1") [{"z2":1, "x":0}] = [0.1, 0.9]
m.cpt("z1") [{"z2":1, "x":1}] = [0.15, 0.85]
```

```
m.cpt("y") [{"z1":0, "z3":0}] = [0.5, 0.5]
m.cpt("y") [{"z1":0, "z3":1}] = [0.45, 0.55]
m.cpt("y") [{"z1":1, "z3":0}] = [0.4, 0.6]
m.cpt("y") [{"z1":1, "z3":1}] = [0.35, 0.65]

d = csl.CausalModel(m, [("X-Z2", ["x", "z2"]),
                        ("X-Z3", ["x", "z3"]),
                        ("X-Y", ["x", "y"]),
                        ("Y-Z2", ["y", "z2"])],
                    True)
```

```
cslnb.showCausalModel(d)
```

nbsphinx-code-borderwhite

```
In [12]: try:
        formula, result, msg = csl.causalImpact(d, on={"y", "z2", "z1", "z3"}, doing={"x"})
    except csl.HedgeException as h:
        print(h.message)

    print(msg)
    display(Math(formula.toLatex()))
```

Do-calculus computations

$$P(z1, y, z3, z2 \mid \hookrightarrow x) = P(z3 \mid z2) \cdot P(z1 \mid x, z2) \cdot \frac{\sum_{x'} P(y \mid x', z1, z2, z3) \cdot P(x' \mid z2) \cdot P(z3 \mid x', z2) \cdot P(z2)}{\sum_{x', y'} P(y' \mid x', z1, z2, z3) \cdot P(x' \mid z2) \cdot P(z3 \mid x', z2) \cdot P(z2)} \cdot P(z2)$$

nbsphinx-code-borderwhite



In [13]: *# computation for this formula directly in pyAgrum*

```
f1=m.cpt("x")*m.cpt("z2")*m.cpt("z3")*m.cpt("y")
f2=f1.margSumOut(["x"])
f3=f1.margSumOut(["x","y"])
f4=f2/f3
pyResult=m.cpt("z3")*m.cpt("z1")*m.cpt("z2")*f4
```

In [14]: *# computation for this formula directly by creating the causal AST*

```
a = csl.ASTposteriorProba(m,{"z1"},{"x","z2"})
b= csl.ASTposteriorProba(m,{"y","z3"},{"x","z1","z2"})
c = csl.ASTjointProba(["x","z2"])
correct = csl.ASTmult(a,csl.ASTsum(["x"],csl.ASTmult(b,c)))
```

```
print("According to [ref], the result should be :")
display(Math(correct.toLatex()))
```

According to [ref], the result should be :

$$P(z_1 | x, z_2) \cdot \left( \sum_x P(y, z_3 | z_1, z_2) \cdot P(x, z_2) \right)$$

In [15]: *# computation for that formula*

```
ie=gum.LazyPropagation(m)
refResult=(ie.evidenceJointImpact(["y","z3"],["x","z1","z2"])*
            ie.evidenceJointImpact(["x","z2"],[])
            ).margSumOut(["x"])* m.cpt("z1")
```

In [16]: `print("Maximum error between these 3 versions : {}".format(max((refResult-pyResult).`

`↪abs().max(),`

`(refResult-result).`

`↪abs().max(),`

`(pyResult-result).new_`

`↪abs().max())))`

Maximum error between these 3 versions : 5.551115123125783e-17

## Unidentifiability

In [17]: `m1 = gum.fastBN("z1->x->z2->y")`

```
cdg = csl.CausalModel(m1, [("Z1X",["z1","x"]),
                           ("Z1Y",["z1","y"]),
                           ("Z1-Z1",["z1","z2"]),
                           ("XY",["x","y"])
                           ], True)
```

```
cslnb.showCausalModel(cdg)
```

nbsphinx-code-borderwhite

In [18]: `err = cslnb.showCausalImpact(cdg,"y","x",values={"x":0})`

<IPython.core.display.HTML object>

**another one**

In [19]: `# EXEMPLE PAGE 17 : http://ftp.cs.ucla.edu/pub/stat\_ser/r350.pdf`

```
m1 = gum.BayesNet()
m1.add(gum.LabelizedVariable("x"))
m1.add(gum.LabelizedVariable("y"))
m1.add(gum.LabelizedVariable("z1"))
m1.add(gum.LabelizedVariable("z2"))
m1.add(gum.LabelizedVariable("z3"))

m1.addArc(2,4)
m1.addArc(2,0)
m1.addArc(3,4)
m1.addArc(3,1)
m1.addArc(4,1)
m1.addArc(4,0)
m1.addArc(0,1)

gnb.showBN(m1)
d = cs1.CausalModel(m1)
nbsphinx-code-borderwhite
```

In [20]: `display(Math(cs1.identifyingIntervention(d,{"z1","z2","z3","y"}, {"x"}).toLatex()))`  
nbsphinx-code-borderwhite
$$P(z_3 | z_1, z_2) \cdot P(z_2) \cdot P(z_1) \cdot P(y | x, z_2, z_3)$$

In [21]: `display(Math(cs1.identifyingIntervention(d,{"y"}, {"x"}).toLatex()))`  
nbsphinx-code-borderwhite
$$\sum_{z_1, z_2, z_3} P(z_3 | z_1, z_2) \cdot P(z_2) \cdot P(z_1) \cdot P(y | x, z_2, z_3)$$

In [22]: `display(Math(cs1.identifyingIntervention(d,{"z1","z3","y"}, {"x","z2"}).toLatex()))`  
nbsphinx-code-borderwhite
$$P(z_3 | z_1, z_2) \cdot P(z_1) \cdot P(y | x, z_2, z_3)$$

In [23]: `display(Math(cs1.identifyingIntervention(d,{"y"}, {"x","z2"}).toLatex()))`  
nbsphinx-code-borderwhite
$$\sum_{z_1, z_3} P(z_3 | z_1, z_2) \cdot P(z_1) \cdot P(y | x, z_2, z_3)$$

**Other example**

In [24]: `#http://www.stats.ox.ac.uk/~lienart/gml15-causalinference.html`

```
m1 = gum.BayesNet()
m1.add(gum.LabelizedVariable("a"))
m1.add(gum.LabelizedVariable("p"))
m1.add(gum.LabelizedVariable("b"))
m1.add(gum.LabelizedVariable("y"))

m1.addArc(0,1)
m1.addArc(1,2)
m1.addArc(0,3)
m1.addArc(1,3)
m1.addArc(2,3)
gnb.showBN(m1)
d = cs1.CausalModel(m1)
```

nbsphinx-code-borderwhite

```
In [25]: display(Math(csl.identifyingIntervention(d,{"y"}, {"a","b"}).toLatex()))
```

$$\sum_p P(y \mid a, b, p) \cdot P(p \mid a)$$

nbsphinx-code-borderwhite

### example f

```
In [26]: #https://cse.sc.edu/~mgv/talks/AIM2010.ppt , example (f)
```

```
m1 = gum.BayesNet()
m1.add(gum.LabelizedVariable("X"))
m1.add(gum.LabelizedVariable("Y"))
m1.add(gum.LabelizedVariable("Z1"))
m1.add(gum.LabelizedVariable("Z2"))

m1.addArc(0,1)
m1.addArc(0,2)
m1.addArc(2,3)
m1.addArc(3,1)
m1.addArc(2,1)
d = csl.CausalModel(m1, [("I1",["X","Z1"]), ("I2",["Y","Z1"])],True)
cslnb.showCausalModel(d)
```

nbsphinx-code-borderwhite

```
In [27]: try:
        display(Math(csl.identifyingIntervention (d,{"Y"}, {"X"}).toLatex()))
except csl.HedgeException as e:
    print("Hedge exception : {}".format(e))
```

Hedge exception : Hedge Error: G={'Y', 'Z1', 'X'}, G[S]={'Z1', 'Y'}

### Example [Pearl,2009] Causality, p66

```
In [28]: bn = gum.fastBN("Z1->Z2->Z3->Y<-X->Z2;Z2->Y;Z1->X->Z3<-Z1")
gnb.showBN(bn)
```

nbsphinx-code-borderwhite

```
In [29]: c = csl.CausalModel(bn, [("Z0", ("X", "Z1", "Z3"))], False)
cslnb.showCausalModel(c)
```



nbsphinx-code-borderwhite

```
In [30]: formula, impact, explanation = csl.causalImpact(c, "Y", "X")
cslnb.showCausalImpact(c,"Y","X")
```

<IPython.core.display.HTML object>

In [ ]:

### 1.25.5 Counterfactual : the Effect of Education and Experience on Salary

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org"> (https://agrums.org)</a>	<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html"> (https://agrums.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

This notebook follows the example from “The Book Of Why” (Pearl, 2018) chapter 8 page 251.

#### Counterfactuals

```
In [1]: from IPython.display import display, Math, Latex, HTML

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.causal as csl
import pyAgrum.causal.notebook as cslnb
import os
import math
import numpy as np
import scipy.stats
```

In this example we are interested in the effect of experience and education on the salary of an employee, we are in possession of the following data:

```
<tr>
  <th>Employé</th> <th>EX(u)</th> <th>ED(u)</th> <th> $S_{0}(u)$</th> <th> $S_{1}(u)$</th> <th> $S_{2}(u)$</th>
</tr>
<tr>
  <td>Alice</td> <td>8</td> <td>0</td> <td>86,000</td> <td>?</td> <td>?</td>
</tr>
<tr>
  <td>Bert</td> <td>9</td> <td>1</td> <td>?</td> <td>92,500</td> <td>?</td>
</tr>
<tr>
  <td>Caroline</td> <td>9</td> <td>2</td> <td>?</td> <td>?</td> <td>97,000</td>
</tr>
<tr>
  <td>David</td> <td>8</td> <td>1</td> <td>?</td> <td>91,000</td> <td>?</td>
</tr>
<tr>
  <td>Ernest</td> <td>12</td> <td>1</td> <td>?</td> <td>100,000</td> <td>?</td>
</tr>
<tr>
  <td>Frances</td> <td>13</td> <td>0</td> <td>97,000</td> <td>?</td> <td>?</td>
</tr>
<tr>
```

(continues on next page)

(continued from previous page)

```
<td>etc</td> <td> </td> <td> </td> <td> </td> <td> </td> <td> </td>
</tr>
```

- $EX(u)$  : years of experience of employee  $u$ .  $[0,20]$
- $ED(u)$  : Level of education of employee  $u$  (0:high school degree (low), 1:college degree (medium), 2:graduate degree (high))  $[0,2]$
- $S_i(u)$   $[65k,150k]$  :
- salary (observable) of employee  $u$  if  $i = ED(u)$ ,
- Potential outcome (unobservable) if  $i \neq ED(u)$ , salary of employee  $u$  if he had a level of education of  $i$ .

We are left with the previous data and we want to answer the counterfactual question What would Alice's salary be if she attended college ? (i.e.  $S_1(Alice)$ )

### We create the causal diagram

In this model it is assumed that an employee's salary is determined by his level of education and his experience. Years of experience are also affected by the level of education. Having a higher level of education means spending more time studying hence less experience.

```
In [2]: edex = gum.fastBN("Ux[-2,10]->experience[0,20]<-education{low|medium|high}->salary[65,
    ↪150];"
        "experience->salary<-Us[0,25]")
edex
```

```
Out[2]: (pyAgrum.BayesNet<double>@0000001EA84D50440) BN{nodes: 5, arcs: 5, domainSize: 10^6.
    ↪26276, dim: 141729}
```

However counterfactual queries are specific to one datapoint (in our case Alice), we need to add additional variables to our model to allow for individual variations: \*  $Us$  : unobserved variables that affect salary. $[0,25k]$  \*  $Ux$  : unobserved variables that affect experience. $[-2,10]$

```
In [3]: # no prior information about the individual (datapoint)
edex.cpt("Us").fillWith(1).normalize()
edex.cpt("Ux").fillWith(1).normalize()
# education level(supposed)
edex.cpt("education")[:] = [0.4, 0.4, 0.2]
```

```
In [4]: # To have probabilistic results, we add a perturbation. (Gaussian around the exact
    ↪values)
# we calculate a gaussian distribution
x_min = 0.0
x_max = 4.0

mean = 2.0
std = 0.65

x = np.linspace(x_min, x_max, 5)

y = scipy.stats.norm.pdf(x,mean,std)
print("We'll use the following distribution \n",y)

We'll use the following distribution
[0.00539715 0.18794845 0.61375735 0.18794845 0.00539715]
```



```
In [6]: edex.cpt("salary").fillWithFunction("round(65+2.51*experience+5*education+Us)",
↪noise=list(y))
gnb.showInference(edex)
nbsphinx-code-borderwhite
```

To answer this counterfactual question we will follow the three steps algorithm from “The Book Of Why” (Pearl 2018) chapter 8 page 253 :

### Step 1 : Abduction

Use the data to retrieve all the information that characterizes Alice

From the data we can retrieve Alice’s profile :  $* Ed(Alice) : 0 * Ex(Alice) : 8 * S_0(Alice) : 86k$

We will use Alice’s profile to get  $U_s$  and  $U_x$ , which tell Alice apart from the rest of the data.

```
In [7]: ie=gum.LazyPropagation(edex)
ie.setEvidence({'experience':8, 'education': 'low', 'salary' : "86"})
ie.makeInference()
newUs = ie.posterior("Us")
newUs
```

```
Out[7]: (pyAgrum.Potential<double>@0000001EAA5565340)
Us
↪
↪
↪
↪
0      | 1      | 2      | 3      | 4      | 5      | 6      | 7      | 8
↪  | 9      | 10     | 11     | 12     | 13     | 14     | 15     | 16
↪ | 17     | 18     | 19     | 20     | 21     | 22     | 23     | 24     | 25
↪
↪-----|-----|-----|-----|-----|-----|-----|-----|-----
↪---|-----|-----|-----|-----|-----|-----|-----|-----
↪|-----|-----|-----|-----|-----|-----|-----|-----|-----
↪-----|
0.1889 | 0.6168 | 0.1889 | 0.0054 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.
↪0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.
↪0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.
↪0000 | 0.0000 |
```

```
In [8]: ie=gum.LazyPropagation(edex)
ie.setEvidence({'experience':8, 'education': 'low', 'salary' : "86"})
ie.makeInference()
newUx = ie.posterior("Ux")
newUx
```

```
Out[8]: (pyAgrum.Potential<double>@0000001EAA55648E0)
Ux
↪
↪
↪
↪
-2      | -1      | 0      | 1      | 2      | 3      | 4      | 5      | 6
↪  | 7      | 8      | 9      | 10     |
↪-----|-----|-----|-----|-----|-----|-----|-----|-----
↪---|-----|-----|-----|-----|
0.7604 | 0.2329 | 0.0067 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.
↪0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
```

```
In [9]: gnb.showInference(edex, evs={'experience':8, 'education': "low", 'salary' : "86"},
      ↪ targets={'Ux', 'Us'})
nbsphinx-code-borderwhite
```

## Step 2 & 3 : Action And Prediction

Change the model to match the hypothesis implied by the query (if she had attended university) and then use the data that characterizes Alice to calculate her salary.

We create a counterfactual world with Alice's idiosyncratic factors, and we operate the intervention:

```
In [10]: # the counterfactual world
edexCounterfactual = gum.BayesNet(edex)

In [11]: # we replace the prior probabilities of idiosyncratic factors with potentials,
      ↪ calculated earlier
edexCounterfactual.cpt("Ux").fillWith(newUx)
edexCounterfactual.cpt("Us").fillWith(newUs)
gnb.showInference(edexCounterfactual, size="10")
print("counterfactual world created")
nbsphinx-code-borderwhite
counterfactual world created
```

```
In [12]: # We operate the intervention
edexModele = cs1.CausalModel(edexCounterfactual)
cslnb.showCausalImpact(edexModele, "salary", doing="education", values={"education":
      ↪ "medium"})

<IPython.core.display.HTML object>
```

Since education has no parents in our model (no graph surgery, no causes to emancipate it from), an intervention is equivalent to an observation, the only thing we need to do is to set the value of education:

```
In [13]: gnb.showInference(edexCounterfactual, targets={"salary", 'experience'}, evs={'education':
      ↪ "medium"}, size="10")
nbsphinx-code-borderwhite
```

The result (salary if she had attended college) is given by the formula:

$$\sum_{salary} salary \times P(salary^* \mid RealSalary = 86k, education = 0, experience = 8, education^* = 1)$$

Where variables marked with an asterisk are inobservable.

```
In [14]: formula, adj, exp = cs1.causalImpact(edexModele, "salary", doing="education", values={
      ↪ "education": "medium"})
gnb.showProba(adj)
nbsphinx-code-borderwhite
```

```
In [15]: i = gum.Instantiation(adj)
i.setFirst()
mean = 0
while (not i.end()):
    v = i.val(0)
    mean = mean + (v+65)*adj.get(i)
    i.inc()
print(mean)
```



```
81.84325639929716
```

$$S_1(Alice) = 81k$$

**Alice's salary would be \$81.843 if she had attended college !**

**pyAgrum.causal.counterfactual**

We can now use a function that answers counterfactual queries using the previous algorithm.

```
In [16]: help(csl.counterfactual)

Help on function counterfactual in module pyAgrum.causal._causalImpact:

counterfactual(cm: pyAgrum.causal._CausalModel.CausalModel, profile:
↳ Optional[Dict[str, int]], on: Union[str, Set[str]], whatif: Union[str, Set[str]],
↳ values: Optional[Dict[str, int]] = None) -> 'pyAgrum.Potential'
    Determines the estimation of a counterfactual query following the the three steps
↳ algorithm from "The Book Of Why"
    (Pearl 2018) chapter 8 page 253.

    Determines the estimation of the counterfactual query: Given the "profile"
↳ (dictionary <variable name>:<value>), what
    would variables in "on" (single or list of variables) be if variables in "whatif"
↳ (single or list of variables) had
    been as specified in "values" (dictionary <variable name>:<value>)(optional).

    This is done according to the following algorithm:
    -Step 1-2: compute the twin causal model
    -Step 3 : determine the causal impact of the interventions specified in
↳ "whatif" on the single or list of
    variables "on" in the causal model.

    This function returns the potential calculated in step 3, representing the
↳ probability distribution of "on" given
    the interventions "whatif", if it had been as specified in "values" (if "values"
↳ is omitted, every possible value of
    "whatif")

Parameters
-----
cm: CausalModel
profile: Dict[str,int] default=None
    evidence
on: variable name or variable names set
    the variable(s) of interest
whatif: str|Set[str]
    idiosyncratic nodes
values: Dict[str,int]
    values for certain variables in whatif.

Returns
-----
pyAgrum.Potential
```

(continues on next page)

(continued from previous page)

```
the computed counterfactual impact
```

**Let's try with the previous query :**

```
In [17]: cm_edex= csl.CausalModel(edex)
pot=csl.counterfactual(cm =cm_edex,
                      profile = {'experience':8, 'education': "low", 'salary' : "86"}
                      ↪,
                      whatif={"education"},
                      on={"salary"},
                      values = {"education" : "medium"})
```

```
In [18]: gnb.showProba(pot)
nbsphinx-code-borderwhite
```

We get the same result !

**If we omit values:**

We get every potential outcome :

```
In [19]: pot=csl.counterfactual(cm =cm_edex,
                      profile = {'experience':8, 'education': 'low', 'salary' : '86'}
                      ↪,
                      whatif={"education"},
                      on={"salary"})
```

```
In [20]: gnb.showPotential(pot)
<IPython.core.display.HTML object>
```

**What would Alice's salary be if she had attended college and had 8 years of experience ?**

```
In [21]: pot=csl.counterfactual(cm =cm_edex,
                      profile = {'experience':8, 'education': 'low', 'salary' : '86'}
                      ↪,
                      whatif={"education", "experience"},
                      on={"salary"},
                      values = {"education" : 'medium', "experience" : 8})
```

```
In [22]: gnb.showProba(pot)
nbsphinx-code-borderwhite
```

**if she attended college and had 8 years of experience Alice's salary would be 91k !**

In the previous query, Alice's salary if she attended college was lower than her actual salary, that's because in the counterfactual world where she attended college she had less time to work hence her diminished salary.

In this query, Alice's counterfactual salary was higher than her actual salary (+5k corresponding to one level of education), that's because in the counterfactual world Alice attended college and still had time to work 8 years, so her salary went up.

**if she had more experience :**

of course, her salary goes up.

```
In [23]: pot=cs1.counterfactual(cm =cm_edex,
    profile = {'experience':8, 'education': 'low', 'salary' : '86'}
    ↪,
    whatif={"education", "experience"},
    on={"salary"},
    values = {"education" : 'medium', "experience" : 10})
gnb.showProba(pot)
nbsphinx-code-borderwhite
```

```
In [24]: twin=cs1.counterfactualModel(cm = cs1.CausalModel(edex),
    profile = {'experience':8, 'education': 'low', 'salary' : '86'}
    ↪,
    whatif={"experience"})
gnb.showInference(twin.observationalBN(),size="10")
nbsphinx-code-borderwhite
```

```
In [25]: edexModeleWithout = cs1.CausalModel(edex) #(<latent variable name>, <list of affected_
    ↪variables' ids>).
edexModeleWithout
```

```
Out[25]: <pyAgrum.causal._CausalModel.CausalModel at 0x1eaa631f820>
```

**Let's try with the previous queries :**

```
In [26]: pot = cs1.counterfactual(cm = edexModeleWithout,
    profile = {'experience':8, 'education': "low", 'salary' : "86"
    ↪},
    whatif={"education"},
    on={"salary"},
    values = {"education" : "medium"})
gnb.showProba(pot)
nbsphinx-code-borderwhite
```

```
In [27]: pot=cs1.counterfactual(cm = edexModeleWithout,
    profile = {'experience':8, 'education': 'low', 'salary' : '86'}
    ↪,
    whatif={"education", "experience"},
    on={"salary"},
    values = {"education" : 'medium', "experience" : 8})
gnb.showProba(pot)
```

nbsphinx-code-borderwhite

We get the same results.

**Latent variable between  $U_x$  and *experience* :**

```
In [28]: edexModelWithOne = cs1.CausalModel(edex, [("u1", ["Ux", "experience"])], False) #(  
    ↪ <latent variable name>, <list of affected variables' ids>).  
edexModelWithOne
```

```
Out[28]: <pyAgrum.causal._CausalModel.CausalModel at 0x1eaa51ed390>
```

```
In [29]: pot = cs1.counterfactual(cm = edexModelWithOne,  
    ↪ profile = {'experience':8, 'education': "low", 'salary' : "86"  
    ↪ },  
    whatif={"education"},  
    on={"salary"},  
    values = {"education" : "medium"})  
gnb.showProba(pot)  
nbsphinx-code-borderwhite
```

With one latent variable between  $U_x$  and *experience*, we get \$96k corresponding to one education level (we don't need to worry about experience any more.)

```
In [30]: pot = cs1.counterfactual(cm = edexModelWithOne,  
    ↪ profile = {'experience':8, 'education': "low", 'salary' : "86"  
    ↪ },  
    whatif={"education"},  
    on={"salary"},  
    values = {"education" : "high"})  
gnb.showProba(pot)  
nbsphinx-code-borderwhite
```

```
In [ ]:
```

## 1.26 Examples

### 1.26.1 Asthma



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: import matplotlib.pyplot as plt  
  
# import the computation tools of aGrUM  
import pyAgrum as gum
```

(continues on next page)

(continued from previous page)

```
# import the graphical display functions
import pyAgrum.lib.notebook as gnb
```

## The model

```
In [2]: # load the "asthma" Bayesian network
bn=gum.loadBN('res/asthma.bif')
```

```
In [3]: # display the Bayesian network
gnb.showBN(bn,nodeColor={n:0.9 for n in bn.names()},cmap=plt.get_cmap('Blues'))
nbsphinx-code-borderwhite
```

```
In [4]: # display the conditional probability table of asthma given pollution
gnb.showPotential(bn.cpt(bn.idFromName('asthma')),digits=4)

<IPython.core.display.HTML object>
```

## Some inference

```
In [5]: # display the probability distribution of Variable "traffic"
gnb.showPosterior (bn, {}, "traffic" )
nbsphinx-code-borderwhite
```

```
In [6]: # display the probability distribution of Variable "pollution"
gnb.showPosterior ( bn, {}, "pollution")
nbsphinx-code-borderwhite
```

```
In [7]: # display the distribution somewhat differently
gum.getPosterior ( bn, {}, "pollution")
```

```
Out[7]: (pyAgrum.Potential<double>@00000222A63E67A0)
pollution
```

	1	2	3	4	5	6	7	8	9
10	0.0000	0.0355	0.4233	0.2788	0.1475	0.0629	0.0164	0.0219	0.0082
0082	0.0000	0.0055							

```
In [8]: # more interesting: display the posterior distribution of "asthma"
# given that we observed that time is 8:00 and weather is cloudy
gnb.showPosterior (bn, {'hour' : 8, 'weather' : 'cloudy'}, "asthma" )
nbsphinx-code-borderwhite
```

```
In [9]: # show the complete model on morning (from 8 to 12)
gnb.showInference(bn, evs={'hour' : [0]*8+[1]*5+[0]*11})
nbsphinx-code-borderwhite
```

```
In [10]: # show the posterior distributions of all the variables given that
# we observed that heure=8 and meteo=nuageux.
# the tables in beige represent the observations
gnb.flow.row(gnb.getInference(bn,size="9",evs={'hour' : 8, 'weather' : 'cloudy'}),
```

(continues on next page)



(continued from previous page)

```

gnb.getInference(bn,size="9",evs={'hour': 7, 'accident' : 'yes'}),
captions=["When time is 8:00 and weatcher is cloudy","When time is 7:00.
↪and there hase been an accident"])
<IPython.core.display.HTML object>

```

## 1.26.2 Kaggle Titanic

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org"> (https://agrum.org)</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

```

In [1]: import pandas
import os
import math
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
from pyAgrum.lib.bn2roc import showROC_PR

from sklearn.metrics import accuracy_score, roc_auc_score, confusion_matrix
import pandas as pd

```

### Titanic: Machine Learning from Disaster

This notebook is an introduction to the [Kaggle titanic challenge](https://www.kaggle.com/c/titanic) (<https://www.kaggle.com/c/titanic>). The goal here is not to produce the best possible classifier, at least not yet, but to show how pyAgrum and Bayesian networks can be used to easily and quickly explore and understand data.

To undstand this notebook, basic knowledge of Bayesian networks is required. If you are looking for an introduction to pyAgrum, check [this notebook](http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/Tutorial.ipynb.html) (<http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/Tutorial.ipynb.html>).

This notebook present three different Bayesien Networks techniques to answer the Kaggle Titanic challenge. The first approach we will answer the challenge without using the training set and we will only use our prior knowledge about shipwrecks. In the second approach we will only use the training set with pyAgrum's machine learning algorithms. Finally, in the third approach we will use both prior knowledge about shipwrecks and machine learning.

Before we start, some disclaimers about aGrUM and pyAgrum.

**aGrUM** is a C++ library designed for easily building applications using graphical models such as Bayesian networks, influence diagrams, decision trees or Markov decision processes.

**pyAgrum** is a Python wrapper for the C++ aGrUM library. It provides a high-level interface to the part of aGrUM allowing to create, handle and make computations into Bayesian networks. The module mainly is an application of the **SWIG** (<http://www.swig.org/>) interface generator. Custom-written code is added to simplify and extend the aGrUM API.

Both projects are [open source](https://gitlab.com/agrumery/aGrUM/blob/master/LICENSE) (<https://gitlab.com/agrumery/aGrUM/blob/master/LICENSE>) and can be freely downloaded from **aGrUM's gitlab repository** (<https://gitlab.com/agrumery/aGrUM>) or installed using **pip** or **anaconda** (<http://agrum.gitlab.io/pages/pyagrum.html>).

If you have questions, remarks or suggestions, feel free to ask us on [info@agrum.org](mailto:info@agrum.org).

## Pretreatment

We will be using `pandas` (<http://pandas.pydata.org/>) to setup the learning data to fit with pyAgrum requirements.

```
In [2]: traindf=pandas.read_csv('res/titanic/train.csv')

testdf=pandas.merge(pandas.read_csv('res/titanic/test.csv'),
                    pandas.read_csv('res/titanic/gender_submission.csv'),
                    on="PassengerId")
```

This merges both the test base with the fact that a passager has survived or not.

```
In [3]: for k in traindf.keys():
        print('{0}: {1}'.format(k, len(traindf[k].unique())))
```

```
PassengerId: 891
Survived: 2
Pclass: 3
Name: 891
Sex: 2
Age: 89
SibSp: 7
Parch: 7
Ticket: 681
Fare: 248
Cabin: 148
Embarked: 4
```

Looking at the number of unique values for each variable is necessary since Bayesian networks are discrete models. We will want to reduce the domain size of some discrete variables (like age) and discretize continuous variables (like Fare).

For starters you can filter out variables with a large number of values. Choosing a large number will have an impact on performances, which boils down to how much CPU and RAM you have at your disposal. Here, we choose to filter out any variable with more than 10 different outcomes.

```
In [4]: for k in traindf.keys():
        if len(traindf[k].unique())<=15:
            print(k)
```

```
Survived
Pclass
Sex
SibSp
Parch
Embarked
```

This leaves us with 6 variables, not much but still enough to learn a Bayesian network. Will just add one more variable by reducing the cardinality of the Age variable.

```
In [5]: testdf=pandas.merge(pandas.read_csv('res/titanic/test.csv'),
                          pandas.read_csv('res/titanic/gender_submission.csv'),
                          on="PassengerId")

def forAge(row):
    try:
        age = float(row['Age'])
        if age < 1:
```

(continues on next page)

(continued from previous page)

```

        #return '[0;1['
        return 'baby'
    elif age < 6:
        #return '[1;6['
        return 'toddler'
    elif age < 12:
        #return '[6;12['
        return 'kid'
    elif age < 21:
        #return '[12;21['
        return 'teen'
    elif age < 80:
        #return '[21;80['
        return 'adult'
    else:
        #return '[80;200]'
        return 'old'
except ValueError:
    return np.nan

def forBoolean(row, col):
    try:
        val = int(row[col])
        if row[col] >= 1:
            return "True"
        else:
            return "False"
    except ValueError:
        return "False"

def forGender(row):
    if row['Sex'] == "male":
        return "Male"
    else:
        return "Female"

```

testdf

```

Out[5]:
  PassengerId  Survived  Pclass
0            892         0      3
1            893         0      3
2            894         0      2
3            895         0      3
4            896         0      3
..          ...         ...   ...
413          1305         0      3
414          1306         1      1
415          1307         0      3
416          1308         0      3
417          1309         0      3

  Name \
0      Kelly, Mr. James
1      Wilkes, Mrs. James (Ellen Needs)
2      Myles, Mr. Thomas Francis
3      Wirz, Mr. Albert
4      Hirvonen, Mrs. Alexander (Helga E Lindqvist)
..      ...
413     Spector, Mr. Woolf
414     Oliva y Ocana, Dona. Fermina
415     Saether, Mr. Simon Sivertsen
416     Ware, Mr. Frederick
417     Peter, Master. Michael J

  Sex  Age  SibSp  Parch  Ticket  Fare  Cabin  Embarked \
0  male  34.5     0     0   330911   7.8292   NaN      Q
1  female  47.0     1     0   363272   7.0000   NaN      S
2  male   62.0     0     0   240276   9.6875   NaN      Q
3  male   27.0     0     0   315154   8.6625   NaN      S

```

(continues on next page)



(continued from previous page)

```

4      female  22.0      1      1      3101298  12.2875  NaN      S
..      ...      ...      ...      ...      ...      ...      ...
413     male   NaN      0      0      A.5. 3236   8.0500  NaN      S
414  female  39.0      0      0      PC 17758  108.9000  C105     C
415     male  38.5      0      0  SOTON/O.Q. 3101262  7.2500  NaN      S
416     male   NaN      0      0      359309   8.0500  NaN      S
417     male   NaN      1      1      2668    22.3583  NaN      C

      Survived
0          0
1          1
2          0
3          0
4          1
..      ...
413         0
414         1
415         0
416         0
417         0

[418 rows x 12 columns]

```

When pretreating data, you will want to wrap your changes inside a function, this will help you keep track of your changes and easily compare them.

```

In [6]: def pretreat(df):
        if 'Survived' in df.columns:
            df['Survived'] = df.apply(lambda row: forBoolean(row, 'Survived'), axis=1)
        df['Age'] = df.apply(forAge, axis=1)
        df['SibSp'] = df.apply(lambda row: forBoolean(row, 'SibSp'), axis=1)
        df['Parch'] = df.apply(lambda row: forBoolean(row, 'Parch'), axis=1)
        df['Sex'] = df.apply(forGender, axis=1)
        dropped_cols = [col for col in ['PassengerId', 'Name', 'Ticket', 'Fare', 'Cabin']]
        if col in df.columns:
            df = df.drop(dropped_cols, axis=1)
            df = df.rename(index=str, columns={'Sex': 'Gender', 'SibSp': 'Siblings', 'Parch':
            'Parents'})
            df.dropna(inplace=True)
        return df

traindf = pandas.read_csv('res/titanic/train.csv')
testdf = pandas.merge(pandas.read_csv('res/titanic/test.csv'),
                      pandas.read_csv('res/titanic/gender_submission.csv'),
                      on="PassengerId")

traindf = pretreat(traindf)
testdf = pretreat(testdf)

```

We will need to save this intermediate learning database, since pyAgrum accepts only files as inputs. As a rule of thumb, save your CSV using comma as separators and do not quote values when you plan to use them with pyAgrum.

```

In [7]: import csv
        traindf.to_csv('res/titanic/post_train.csv', index=False)
        testdf.to_csv('res/titanic/post_test.csv', index=False)

```

## Modeling without learning

In some cases, we might not have any data to learn from. In such cases, we can rely on experts to provide correlation between variables and conditional probabilities.

It can be simpler to start with a simple topography, leaving room to add more complex correlations as the model is confronted against data. Here, we will use three hypotheses: - All variables are independent conditionally to each other given the fact that a passenger has survived or not. - Women and children are more likely to survive. - The more sibling or parents aboard, the less likely the passenger will survive.

The first assumption results in the following DAG for our Bayesian network:

```
In [8]: bn = gum.BayesNet("Surviving Titanic")
bn = gum.fastBN("Age{baby|toddler|kid|teen|adult|old}<-Survived{False|True}->Gender
->{Female|Male};Siblings{False|True}<-Survived->Parents{False|True}")
print(bn.variable("Survived"))
print(bn.variable("Age"))
print(bn.variable("Gender"))
print(bn.variable("Siblings"))
print(bn.variable("Parents"))
```

bn

```
Survived:Labelized({False|True})
Age:Labelized({baby|toddler|kid|teen|adult|old})
Gender:Labelized({Female|Male})
Siblings:Labelized({False|True})
Parents:Labelized({False|True})
```

```
Out[8]: (pyAgrum.BayesNet<double>@00000015731AFF180) BN{nodes: 5, arcs: 4, domainSize: 96, dim:
-> 26}
```

Hypothesis two and three can help us define the parameters for this Bayesian network. Remember that we assume that we do not have any data to learn from. So we will use simple definition such as “a women is 10 times more likely to survive than a man”. We can then normalize the values to obtain a proper conditional probability distribution.

This technique may not be the most precise or scientifically sounded, it however has the advantage to be easy to use.

```
In [9]: bn.cpt('Survived')[:] = [100, 1]
bn.cpt('Survived').normalizeAsCPT()
bn.cpt('Survived')
```

```
Out[9]: (pyAgrum.Potential<double>@00000015731815A20)
```

	Survived	
False	True	
-----	-----	
0.9901	0.0099	

```
In [10]: bn.cpt('Age')[{'Survived':0}] = [ 1, 1, 1, 10, 10, 1]
bn.cpt('Age')[{'Survived':1}] = [ 10, 10, 10, 1, 1, 10]
bn.cpt('Age').normalizeAsCPT()
bn.cpt('Age')
```

```
Out[10]: (pyAgrum.Potential<double>@00000015731815B00)
```

	Age	
Surviv	baby  toddler  kid  teen  adult  old	
-----	----- ----- ----- ----- -----	
False	0.0417   0.0417   0.0417   0.4167   0.4167   0.0417	
True	0.2381   0.2381   0.2381   0.0238   0.0238   0.2381	

```
In [11]: bn.cpt('Gender')[{'Survived':0}] = [ 1, 1]
bn.cpt('Gender')[{'Survived':1}] = [ 10, 1]
bn.cpt('Gender').normalizeAsCPT()
bn.cpt('Gender')
```

```
Out[11]: (pyAgrum.Potential<double>@00000015731815A00)
      || Gender      |
Surviv||Female      |Male      |
-----||-----|-----|
False || 0.5000 | 0.5000 |
True  || 0.9091 | 0.0909 |
```

```
In [12]: bn.cpt('Siblings')[{'Survived':0}] = [ 1, 10]
bn.cpt('Siblings')[{'Survived':1}] = [ 10, 1]
bn.cpt('Siblings').normalizeAsCPT()
bn.cpt('Siblings')
```

```
Out[12]: (pyAgrum.Potential<double>@000000157318159C0)
      || Siblings    |
Surviv||False      |True      |
-----||-----|-----|
False || 0.0909 | 0.9091 |
True  || 0.9091 | 0.0909 |
```

```
In [13]: bn.cpt('Parents')[{'Survived':0}] = [ 1, 10]
bn.cpt('Parents')[{'Survived':1}] = [ 10, 1]
bn.cpt('Parents').normalizeAsCPT()
bn.cpt('Parents')
```

```
Out[13]: (pyAgrum.Potential<double>@00000015731815740)
      || Parents     |
Surviv||False      |True      |
-----||-----|-----|
False || 0.0909 | 0.9091 |
True  || 0.9091 | 0.0909 |
```

Now we can start using the Bayesian network and check that our hypothesis hold.

```
In [14]: gnb.showInference(bn,size="10")
nbsphinx-code-borderwhite
```

We can see here that most passengers (99% of them) will not survive and that we have almost as much women (50.4%) as men (49.6%). The majority of passengers are either teenagers or adults. Finally, most passenger had siblings or parents aboard.

Recall that we have not use any data to learn the Bayesian Netork's parameters and our expert did not have any knowledge about the passengers aboard the Titanic.

```
In [15]: gnb.showInference(bn,size="10", evs={'Survived':'False'})
gnb.showInference(bn,size="10", evs={'Survived':'True'})
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
```

Here, we can see that our second and third hypothesis hold since when we enter evidendce that a passenger survived, it is more likely to be a woman with no siblings or parents. On the contrary, if we observe that a passenger did not survive we can see that it is more likely to be a man with siblings or parents.

```
In [16]: gnb.showInference(bn,size="10", evs={'Survived':'True', 'Gender':'Male'})
gnb.showInference(bn,size="10", evs={'Gender':'Male'})
```

nbsphinx-code-borderwhite  
nbsphinx-code-borderwhite

This validates our first hypothesis: if we know that a passenger survived or not, then evidence about that passenger does not changes our belief about other variables. On the contrary, if we do not know if a passenger survived, then evidence about the passenger will change our belief about other variables, including the fact that he or she survived or not.

```
In [17]: ie=gum.LazyPropagation(bn)

def init_belief(engine):
    # Initialize evidence
    for var in engine.BN().names():
        if var != 'Survived':
            engine.addEvidence(var, 0)

def update_beliefs(engine, bayesNet, row):
    # Update beliefs from a given row less the Survived variable
    for var in bayesNet.names():
        if var == "Survived":
            continue
        try:
            label = str(row.to_dict()[var])
            idx = bayesNet.variable(var).index(str(row.to_dict()[var]))
            engine.chgEvidence(var, idx)
        except gum.NotFound:
            # this can happend when value is missing is the test base.
            pass
    engine.makeInference()

def is_well_predicted(engine, bayesNet, auc, row):
    update_beliefs(engine, bayesNet, row)
    marginal = engine.posterior('Survived')
    outcome = row.to_dict()['Survived']
    if outcome == "False": # Did not survived
        if marginal.toarray()[1] < auc:
            return "True Positive"
        else:
            return "False Negative"
    else: # Survived
        if marginal.toarray()[1] >= auc:
            return "True Negative"
        else:
            return "False Positive"

init_belief(ie)
ie.addTarget('Survived')
result = testdf.apply(lambda x: is_well_predicted(ie, bn, 0.5, x), axis=1)
result.value_counts(True)
```

```
Out[17]: True Positive      0.516746
False Positive    0.322967
False Negative    0.119617
True Negative     0.040670
dtype: float64
```

```
In [18]: positives = sum(result.map(lambda x: 1 if x.startswith("True") else 0 ))
total = result.count()
```

(continues on next page)

(continued from previous page)

```
print("{0:.2f}% good predictions".format(positives/total*100))
```

```
55.74% good predictions
```

This first model achieve a 55.74% of good predictions, not a good result but we have plenty of room to improve it.

## Pre-learning

We will now learn a Bayesian network from the training set without any prior knowledge about shipwreks.

Before learning a Bayesian network, we first need to create a template. This is not mandatory, however it is sometimes usefull since not all varaibles values are present in the learning base (in this example the number of relatives).

If during the learning step, the algorithm encounters an unknown value it will raise an error. This would be an issue if we wanted to automitize our classifier but, we will directly use values working with the test and learning base. This is not ideal but the objective here it to explore the data fast, not thoroughly.

To help creating de the template Bayesian network that we will use to learn our classifier, let us firt recall all the variables wa have at our disposal.

```
In [19]: df = pandas.read_csv('res/titanic/post_train.csv')
for k in traindf.keys():
    print('{0}: {1}'.format(k, len(traindf[k].unique())))
```

```
Survived: 2
Pclass: 3
Gender: 2
Age: 6
Siblings: 2
Parents: 2
Embarked: 3
```

From here, creating the BayesNet is straitforward: for each variable we either use the RangeVariable class or the LabelizedVariable.

The RangeVariable class creates a discrete random variable over a range. With the LabelizedVariable you will need to add each label only by one. Note however that you can pass an argument to create as much labels starting from 0.

```
In [20]: template=gum.BayesNet()
template.add(gum.LabelizedVariable("Survived", "Survived", ['False', 'True']))
template.add(gum.RangeVariable("Pclass", "Pclass",1,3))
template.add(gum.LabelizedVariable("Gender", "The passenger's gender",['Female', 'Male',
↪]))
template.add(gum.LabelizedVariable("Siblings", "Siblings",['False', 'True']))
template.add(gum.LabelizedVariable("Parents", "Parents",['False', 'True']))
template.add(gum.LabelizedVariable("Embarked", "Embarked", ['', 'C', 'Q', 'S']))
template.add(gum.LabelizedVariable("Age", "The passenger's age category", ["baby",
↪"toddler", "kid", "teen", "adult", "old"]))
gnb.showBN(template)
nbsphinx-code-borderwhite
```

You can also let the learning algorithm create the BayesNet random variables. However please be aware that the algorithm will no be able to handle values absent from the learning database.

## Learning a probabilistic model

We can now learn our first Bayesian network. As you will see, this is really easy.

```
In [21]: learner = gum.BNLearner(df, template)
bn = learner.learnBN()
bn

Out[21]: (pyAgrum.BayesNet<double>@00000015731AFE190) BN{nodes: 7, arcs: 9, domainSize: 1152,
↳ dim: 78}
```

In a notebook, a Bayesian network will automatically be shown graphically, you can also use the helper function `gnb.showBN(bn)`.

## Exploring the data

Now that we have a BayesNet, we can start looking how the variables correlate with each other. pyAgrum offer the perfect tool for that: the information graph.

```
In [22]: import pyAgrum.lib.explain as explain
explain.showInformation(bn, {}, size="20")

<IPython.core.display.HTML object>
```

To read this graph, you must understand what the entropy of a variable means: the highest the value the more uncertain the variable marginal probability distribution is (maximum entropy being the equiprobable law). The lowest the value is, the more /certain/ the law is.

A consequence of how entropy is calculated, is that entropy tends to get bigger if the random variable has many modalities.

What the information graph tells us is that the decade variable has a high entropy. Thus, we can conclude that the passengers decade is distributed between all of its modalities.

What it also tells us, it that high modality variables with low entropy, such as Parch or SibSp, are not evenly distributed.

Let us look at the variables marginal probability by using the `showInference()` function.

```
In [23]: gnb.showInference(bn)
nbsphinx-code-borderwhite
```

The `showInference()` is really useful as it shows the marginal probability distribution for each random variable of a BayesNet.

We can now confirm what the entropy learned us: Parch and SibSp are unevenly distributed and decade is more evenly distributed.

Let's focus on the Kaggle challenge now, and look at the Survived variable. We show a single posterior using the `showPosterior()` function.

```
In [24]: gnb.showPosterior(bn, evs={}, target='Survived')
nbsphinx-code-borderwhite
```

So more than 40% of the passenger in our learning database survived.

So how can we use this BayesNet as a classifier? Given a set of evidence, we can infer an updated posterior distribution of the target variable Survived.

Let's look at the odds of surviving as a man in his thirties.

```
In [25]: gnb.showPosterior(bn, evs={"Gender": "Male", "Age": "adult"}, target='Survived')
```

nbsphinx-code-borderwhite

And now the odds of an old lady to survive.

```
In [26]: gnb.showPosterior(bn, evs={"Gender": "Female", "Age": 'old'}, target='Survived')
nbsphinx-code-borderwhite
```

Well, children and ladies first, that's right ?

One last information we will need is which variables are required to predict the Survived variable. To do, we will use the markov blanket of Survived.

```
In [27]: gnb.flow.row(bn, gum.MarkovBlanket(bn, 'Survived'),
                  captions=["Learned Bayesian network", "Markov blanket of 'Survived'"])
<IPython.core.display.HTML object>
```

The Markov Blanket of the Survived variable tells us that we only need to observe Sex and Pclass in order to predict Survived. Not really usefull here but on larger Bayesian networks it can save you a lot of time and CPU.

So how to use this BayesNet we have learned as a classifier ? We simply infer the posterior the Survive variable given the set of evidence we are given, and if the passanger odds of survival are above some value he will be tagged as a survivor.

To compute the best value given the BayesNet and our training database, we can use the showROC() function.

```
In [28]: showROC_PR(bn, 'res/titanic/post_train.csv', 'Survived', 'True', False, True);
nbsphinx-code-borderwhite
```

```
In [29]: ie=gum.LazyPropagation(bn)
init_belief(ie)
ie.addTarget('Survived')
result = testdf.apply(lambda x: is_well_predicted(ie, bn, 0.157935, x), axis=1)
result.value_counts(True)
```

```
Out[29]: True Negative      0.363636
True Positive      0.349282
False Negative     0.287081
dtype: float64
```

```
In [30]: positives = sum(result.map(lambda x: 1 if x.startswith("True") else 0 ))
total = result.count()
print("{0:.2f}% good predictions".format(positives/total*100))

71.29% good predictions
```

With 71% of good prediction, this model performs better than the first one.

## Using BNClassifier

'BNClassifier' is a BN-wrapping object that you can use as a classifier directly. It is inspired and compatible with scikitlearn methods.

```
In [31]: import pyAgrum.skbn as skbn
```

```
In [32]: post_train_df = pandas.read_csv('res/titanic/post_train.csv').dropna().astype(str)
post_test_df = pandas.read_csv('res/titanic/post_test.csv').dropna().astype(str)
```

```
In [33]: targetColumn = 'Survived'

x_train_df = post_train_df.drop(targetColumn, axis=1)
y_train_df = post_train_df[targetColumn]
x_test_df = post_test_df.drop(targetColumn, axis=1)
y_test_df = post_test_df[targetColumn]
```

You create the object. Then you fit the classifier to your training dataset. The fitted classifier can now predict the class for the testing dataset.

```
In [34]: bn = skbn.BNClassifier(prior="NoPrior")
bn.fit(x_train_df, y_train_df)
y_test_pred = bn.predict(x_test_df)

print("{0:.2f}% good predictions".format(accuracy_score(y_test_df, y_test_pred)*100))

86.36% good predictions
```

You can try different parameters and use the same procedure. For example, with a a-priori-smoothing :

```
In [35]: bn_prior = skbn.BNClassifier(prior='Smoothing',priorWeight=1)
bn_prior.fit(x_train_df, y_train_df)
y_test_pred_prior = bn_prior.predict(x_test_df)

print("{0:.2f}% good predictions".format(accuracy_score(y_test_df, y_test_pred_
↪prior)*100))

86.36% good predictions
```

## Making a BN without learning data

In this last part we will combine both methods: we will force the BayesNet DAG and learn its parameters. We will assume the naive bayes hypothesis, which states that all random variables are independant conditionally to the target variable (here the variable Survived).

This results in the following (already seen) above topology.

```
In [36]: bn = gum.BayesNet("Surviving Titanic")
bn = gum.fastBN("Age{baby|toddler|kid|teen|adult|old}<-Survived{False|True}->Gender
↪{Female|Male};Siblings{False|True}<-Survived->Parents{False|True}")
print(bn.variable("Survived"))
print(bn.variable("Age"))
print(bn.variable("Gender"))
print(bn.variable("Siblings"))
print(bn.variable("Parents"))

bn

Survived:Labelized({False|True})
Age:Labelized({baby|toddler|kid|teen|adult|old})
Gender:Labelized({Female|Male})
Siblings:Labelized({False|True})
Parents:Labelized({False|True})

Out[36]: (pyAgrum.BayesNet<double>@00000015731AFF6D0) BN{nodes: 5, arcs: 4, domainSize: 96, dim:
↪ 26}
```

The next step is to learn the parameters, this can easily be done using the learnParameters method.



```
In [37]: learner = gum.BNLearner("res/titanic/post_train.csv", bn)
bn = learner.learnParameters(bn.dag())
gnb.showInference(bn, size="10")
nbsphinx-code-borderwhite
```

If we compare the CPTs obtained here with those defined by our expert in the first example we can see that they differ. They resemble those obtained in the second example. This result is expected since we have learned the parameters from the training data, the learned probabilities distribution should match the data.

The final steps consists of confronting this model against our test dataset.

```
In [38]: showROC_PR(bn, "res/titanic/post_train.csv", 'Survived', "True", True, True)
showROC_PR(bn, "res/titanic/post_test.csv", 'Survived', "True", True, True)

res/titanic/post_train.csv: 100%||
nbsphinx-code-borderwhite
res/titanic/post_test.csv: 100%||
nbsphinx-code-borderwhite
```

```
Out[38]: (0.9879550850811238,
0.47050241859999997,
0.922493887637304,
0.47050241859999997)
```

```
In [39]: ie = gum.LazyPropagation(bn)
init_belief(ie)
ie.addTarget('Survived')
result = testdf.apply(lambda x: is_well_predicted(ie, bn, 0.4705, x), axis=1)
result.value_counts(True)
```

```
Out[39]: True Positive      0.624402
True Negative      0.363636
False Negative      0.011962
dtype: float64
```

```
In [40]: positives = sum(result.map(lambda x: 1 if x.startswith("True") else 0))
total = result.count()
print("{0:.2f}% good predictions".format(positives/total*100))

98.80% good predictions
```



Naive Bayes perform well when used for classification tasks, as shown by the 95% of good predictions achieved by our third model.

## Conclusion

We have demonstrated with different classification techniques using Bayesian networks. In the first approach, we managed to model a classifier without using any training set and relying solely on prior knowledge. In the second approach we used only machine learning techniques. Finally, in the third example we assumed the naive bayes hypothesis and obtained a model combinede

```
In [ ]:
```

### 1.26.3 Naive modeling of credit defaults using a Markov Random Field

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org"> (https://agrum.org)</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

This notebook is the adaptation to pyAgrum of the [model proposed by Gautier Marti](https://marti.ai/quant/2022/01/30/default-distribution-mrf.html) (<https://marti.ai/quant/2022/01/30/default-distribution-mrf.html>) which is itself inspired by the paper [Graphical Models for Correlated Defaults](https://arxiv.org/pdf/0809.1393.pdf) (<https://arxiv.org/pdf/0809.1393.pdf>) (adaptation by **Marvin Lasserre** and Pierre-Henri Wuillemin).

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.mn2graph as m2g
```

#### Constructing the model

Three sectors co-dependent in their stress state are modelled: (Finance, Energy, Retail). For each of these sectors, we consider a universe of three issuers.

Within <i>Finance</i>	Within <i>Energy</i>	Within <i>Retail</i>
Deutsche Bank	EDF	New Look
Metro Bank	Petrobras	Matalan
Barclays	EnQuest	Marks & Spencer

The probability of default of these issuers is partly idiosyncratic and partly depending on the stress within their respective sectors. Some distressed names such as New Look can have a high marginal probability of default, so high that the state of their sector (normal or stressed) does not matter much. Some other high-yield risky names such as Petrobras may strongly depend on how the whole energy sector is doing. On the other hand, a company such as EDF should be quite robust, and would require a very acute and persistent sector stress to move its default probability significantly higher.

We can encode this basic knowledge in terms of potentials:

	Finance	
Energy	no stress	under stress
no stress	90.0000	70.0000
under stress	60.0000	10.0000

Conventions:

Sector variables can be either in the state `no stress` or in the state `under stress` while issuer variables can be either in the state `no default` or in the state `default`.

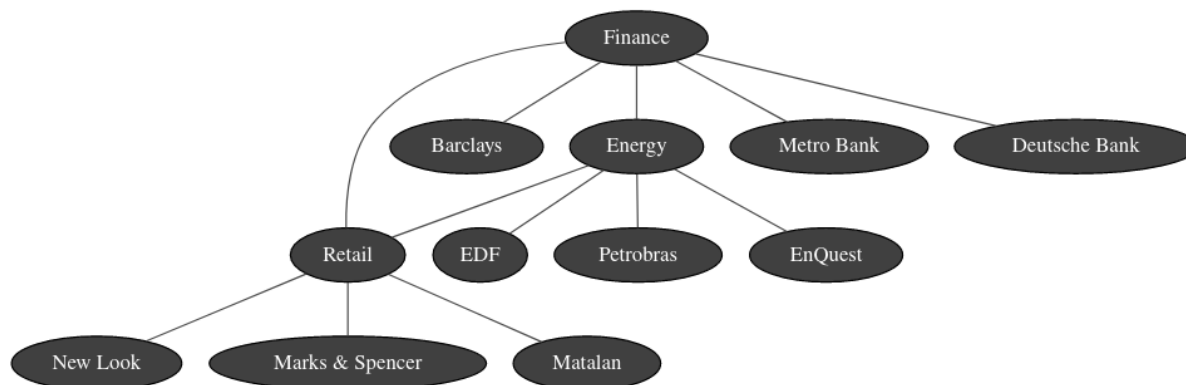
The choice of these potentials is not an easy task; It can be driven by historical fundamental or statistical relationships or forward-looking views.

Given a network structure, and a set of potentials, one can mechanically do inference using the pyAgrum library.

We will show how to obtain a distribution of the number of defaults (and the expected number of defaults), a distribution of the losses (and the expected loss). From the joint probability table, we could also extract the default correlation

tl;dr We use here the library [pyAgrum](https://agrum.org) (<https://agrum.org>) to illustrate, on a toy example, how one can use a Markov Random Field (MRF) for modelling the distribution of defaults and losses in a credit portfolio.

Building accurate PGM/MRF models require expert knowledge for considering a relevant graph structure, and attributing useful potentials. For this toy model, we use the following structure:



The first step consists in creating the model. For that, we use the pyAgrum class `MarkovNet`.

```
In [2]: # building nodes (with types)
mn = gum.MarkovNet('Credit default modeling')

# Adding sector variables
sectors = ['Finance', 'Energy', 'Retail']
finance, energy, retail = [mn.add(gum.LabelizedVariable(name, '', ['no stress',
↪ 'under stress']))
                           for name in sectors]

# Adding issuer variables
edf, petro, eq = [mn.add(gum.LabelizedVariable(name, '', ['no default', 'default']))
                  for name in ['EDF', 'Petrobras', 'EnQuest']]

mb, db, ba = [mn.add(gum.LabelizedVariable(name, '', ['no default', 'default']))
              for name in ['Metro Bank', 'Deutsche Bank', 'Barclays']]

nl, matalan, ms = [mn.add(gum.LabelizedVariable(name, '', ['no default', 'default']))
                   for name in ['New Look', 'Matalan', 'Marks & Spencer']]
```

```
In [3]: # Adding and filling factors between sectors
mn.addFactor([finance, energy][:] = [[90, 70],
                                     [60, 10]]
mn.addFactor([finance, retail][:] = [[80, 10],
                                     [30, 80]]
mn.addFactor([energy, retail][:] = [[60, 5],
                                    [70, 95]]

# Adding factors between sector and issuer
mn.addFactor([db, finance][:] = [[90, 30],
                                 [10, 60]]
mn.addFactor([mb, finance][:] = [[80, 40],
                                 [ 5, 60]]
```

(continues on next page)

(continued from previous page)

```
mn.addFactor([ba, finance][:] = [[90, 20],
                                [20, 50]]

mn.addFactor([edf, energy][:] = [[90, 5],
                                [80, 40]]
mn.addFactor([petro, energy][:] = [[60, 50],
                                [ 5, 60]]
mn.addFactor([eq, energy][:] = [[80, 20],
                                [10, 50]]

mn.addFactor([nl, retail][:] = [[ 5, 60],
                                [ 2, 90]]
mn.addFactor([matalan, retail][:] = [[40, 30],
                                [20, 70]]
mn.addFactor([ms, retail][:] = [[80, 10],
                                [30, 50]]
```

```
In [4]: nodetypes={n:0.1 if n in sectors else
                0.2 for n in mn.names()}

gnb.sideBySide(gnb.getMN(mn,view='graph',nodeColor=nodetypes),
               gnb.getMN(mn,view='factorgraph',nodeColor=nodetypes),
               captions=['The model as a Markov Network','The model as a factor graph
→'])

gnb.sideBySide(mn.factor({'Energy', 'Finance'}),
               mn.factor({'Finance', 'Retail'}),
               mn.factor({'Energy', 'Retail'}),
               mn.factor({'Deutsche Bank', 'Finance'}),
               mn.factor({'Metro Bank', 'Finance'}),
               mn.factor({'Barclays', 'Finance'}),
               mn.factor({'EDF', 'Energy'}),
               mn.factor({'Petrobras', 'Energy'}),
               mn.factor({'EnQuest', 'Energy'}),
               mn.factor({'New Look', 'Retail'}),
               mn.factor({'Matalan', 'Retail'}),
               mn.factor({'Marks & Spencer', 'Retail'}),
               ncols=3)
```

&lt;IPython.core.display.HTML object&gt;

&lt;IPython.core.display.HTML object&gt;

## Making inferences

### Example 1 & 2

The first two examples consists in finding the chance of EDF to default with no evidence and knowing that the energy sector is not under stress. For that, we can use the `getPosterior` method:

```
In [5]: gnb.sideBySide(gum.getPosterior(mn, target='EDF', evs={}),
                     gum.getPosterior(mn, target='EDF', evs={'Energy':'no stress'}),
                     captions=['$P(EDF)$', '$P(EDF|Energy=$under stress)$'])
```

&lt;IPython.core.display.HTML object&gt;

Hence, if we cannot observe the state of the energy sector, then EDF has a higher chance of defaulting (9%) as it is possible that the energy sector is under stress.

### Example 3 & 4

Given that Marks & Spencer (a much safer name) has defaulted, New Look has now an increased chance of defaulting: 97%.

Even if the retail sector is doing ok, New Look is a distressed name with a high probability of default (92%).

```
In [6]: gnb.sideBySide(gum.getPosterior(mn, target='New Look', evs={'Marks & Spencer':
→ 'default'})),
        gum.getPosterior(mn, target='New Look', evs={'Retail': 'no stress'}),
        captions=['P(New Look|Marks & Spencer=default)',
                  'P(New Look | Retail = no stress)'])

<IPython.core.display.HTML object>
```

pyAgrum offers the option to visualize the marginal probability of each variable using `showInference`.

```
In [7]: #gum.config['factorgraph','edge_length_inference']=1.
gnb.showInference(mn, size='7!', nodeColor=nodetypes)
nbsphinx-code-borderwhite
```

### Impact of defaults on a credit portfolio and the distribution of losses

Amongst the 9 available issuers, let's consider the following portfolio containing bonds from these 6 issuers:

```
In [8]: portfolio_names = ['EDF', 'Petrobras', 'EnQuest', 'Matalan', 'Barclays']
portfolio_exposures = [2e6, 400e3, 300e3, 600e3, 3e6]
LGD = 0.9
losses = {portfolio_names[i]:(portfolio_exposures[i]*LGD) for i in
→ range(len(portfolio_names))}

print('** Total notional: USD {} MM'.format(sum(portfolio_exposures)/1e6))

** Total notional: USD 6.3 MM
```

We have a USD 6.3MM total exposure to these names (notional).

We assume that in case of default, we recover only 10% (LGD = 90%).

Let's start using the model:

```
In [9]: ie = gum.ShaferShenoyMNIInference(mn)
ie.makeInference()
p=ie.jointPosterior(set(sectors))

print(f'** There is {100*p[{"Retail":0,"Energy":0,"Finance":0}]:.2f}% chance that no_
→ sector in under stress.')
print()
p

** There is 42.38% chance that no sector in under stress.
```

```
Out [9]: (pyAgrum.Potential<double>@00000021CD885C120)
          || Finance          |
Energy|Retail||no stress|under str|
-----|-----||-----|-----|
```

(continues on next page)

(continued from previous page)

no str	no str	0.4238	0.0083	
under	no str	0.0105	0.0000	
no str	under	0.2999	0.1251	
under	under	0.1215	0.0109	

If we observe that the finance sector is doing ok, and that Marks & Spencer is still doing business as usual, then this is the current joint probabilities associated to the potential defaults in the portfolio:

```
In [10]: ie = gum.ShaferShenoyMNIInference(mn)
ie.addJointTarget(set(portfolio_names))
ie.setEvidence({'Finance': 'no stress', 'Marks & Spencer': 'no default'})
ie.makeInference()
portfolio_posterior = ie.jointPosterior(set(portfolio_names))
```

Note that the line `ie.addJointTarget(set(portfolio_names))` is important here because adding hard evidences removes the corresponding nodes in the graph and then no factor containing the variables in `portfolio_names` can be found. The method `addJointTarget` ensure that such a factor exists.

```
In [11]: portfolio_posterior
```

```
Out[11]: (pyAgrum.Potential<double>@00000021CD885BE00)
          || Barclays          |
Matala|Petrob|EnQues|EDF    ||no default|default |
-----|-----|-----|-----|-----|-----|
no def|no def|no def|no def|| 0.1495 | 0.0332 |
defaul|no def|no def|no def|| 0.1552 | 0.0345 |
no def|defaul|no def|no def|| 0.1268 | 0.0282 |
defaul|defaul|no def|no def|| 0.1350 | 0.0300 |
no def|no def|defaul|no def|| 0.0383 | 0.0085 |
defaul|no def|defaul|no def|| 0.0412 | 0.0092 |
[...4 more line(s) ...]
no def|defaul|no def|defaul|| 0.0081 | 0.0018 |
defaul|defaul|no def|defaul|| 0.0102 | 0.0023 |
no def|no def|defaul|defaul|| 0.0026 | 0.0006 |
defaul|no def|defaul|defaul|| 0.0034 | 0.0008 |
no def|defaul|defaul|defaul|| 0.0077 | 0.0017 |
defaul|defaul|defaul|defaul|| 0.0170 | 0.0038 |
```

We now want to compute the distribution of the number of defaults. For that, we create a new Markov network containing an additional node `Number of defaults` connected to each issuers:

```
In [12]: mn2 = gum.MarkovNet(mn)

mn2.add(gum.RangeVariable('Number of defaults', '', 0, len(portfolio_names)))
factor = mn2.addFactor(['Number of defaults', *portfolio_names]).fillWithFunction('+'.
→ join(portfolio_names))
```

```
In [13]: gum.config['factorgraph', 'edge_length']=1.1
nodetypes={n:0.1 if n in sectors else
            0.3 if "Number of defaults"==n else
            0.2 for n in mn2.names()}
gnb.flow.add(gnb.getMn(mn2, view="graph", size='7!', nodeColor=nodetypes))
gnb.flow.add(gnb.getMn(mn2, view="factorgraph", size='7!', nodeColor=nodetypes))
gnb.flow.display()
```

```
<IPython.core.display.HTML object>
```

Once this new network is created, we can obtain the distribution of `Number of defaults` using `getPosterior`

:

```
In [14]: ndefault_posterior = gum.getPosterior(mn2, target='Number of defaults',
        evs={'Finance': 'no stress', 'Marks &
        ↳ Spencer': 'no default'})
ndefault_posterior
```

```
Out[14]: (pyAgrum.Potential<double>@00000021CD885CFC0)
      Number of defaults
0      | 1      | 2      | 3      | 4      | 5      |
-----|-----|-----|-----|-----|-----|
0.1495 | 0.3620 | 0.3119 | 0.1371 | 0.0357 | 0.0038 |
```

```
In [15]: fig, ax = plt.subplots()
l = ndefault_posterior.tolist()
ax.bar(range(len(l)), l)
ax.set_xlabel('Number of defaults')
ax.set_title('Distribution of the number of defaults in the portfolio')
plt.show()
nbsphinx-code-borderwhite
```

```
In [16]: q=gum.Potential(ndefault_posterior).fillWith(range(ndefault_posterior.domainSize()))
print(f'Expected number of defaults in the portfolio: {(q*ndefault_posterior).sum():.
↳ 1f}')

Expected number of defaults in the portfolio: 1.6
```

Besides the expected number of defaults, the distribution of losses is even more informative:

```
In [17]: pot_losses = gum.Potential(portfolio_posterior)
for i in pot_losses.loopIn():
    pot_losses[i] = sum([losses[key]*i.val(key) for key in losses])
print("Table of losses")
pot_losses
```

Table of losses

```
Out[17]: (pyAgrum.Potential<double>@00000021CD87AD8F0)
      || Barclays
Matala|Petrob|EnQues|EDF ||no defaul|default |
-----|-----|-----|-----|-----|-----|
no def|no def|no def|no def|| 0.0000 | 2700000.0000|
default|no def|no def|no def|| 540000.0000| 3240000.0000|
no def|default|no def|no def|| 360000.0000| 3060000.0000|
default|default|no def|no def|| 900000.0000| 3600000.0000|
no def|no def|default|no def|| 270000.0000| 2970000.0000|
default|no def|default|no def|| 810000.0000| 3510000.0000|
[...4 more line(s) ...]
no def|default|no def|default|| 2160000.0000| 4860000.0000|
default|default|no def|default|| 2700000.0000| 5400000.0000|
no def|no def|default|default|| 2070000.0000| 4770000.0000|
default|no def|default|default|| 2610000.0000| 5310000.0000|
no def|default|default|default|| 2430000.0000| 5130000.0000|
default|default|default|default|| 2970000.0000| 5670000.0000|
```

```
In [18]: expected_loss = (portfolio_posterior * pot_losses ).sum()

print(f'Expected loss with respect to a par value of USD {sum(portfolio_exposures)/
↳ 1e6:.1f} MM is USD {expected_loss / 1e6:.1f} MM.')
```

(continues on next page)

(continued from previous page)

```
print(f'Expected loss: {100 * expected_loss / sum(portfolio_exposures):2.1f}% of the_
↳notional.')
```

Expected loss with respect to a par value of USD 6.3 MM is USD 1.2 MM.  
Expected loss: 18.6% of the notional.

```
In [19]: cuts=list(np.percentile([pot_losses[i] for i in pot_losses.loopIn()],range(0,101,10)))
cuts[0] = cuts[0] - 0.001

loss_bucket_distribution = []
d = gum.Potential(pot_losses)
for j in range(len(cuts) - 1):
    for i in pot_losses.loopIn():
        d[i] = 1 if cuts[j]<pot_losses[i]<=cuts[j+1] else 0
    loss_bucket_distribution.append(((portfolio_posterior*d).sum()))
```

```
In [20]: buckets = [f'({cuts[i]:10.0f}, {cuts[i+1]:10.0f})' for i in range(len(cuts)-1)]

fig, ax = plt.subplots()
ax.bar(range(10), height=loss_bucket_distribution)
ax.set_xticks(range(10), labels=buckets, rotation=90)
ax.set_title('Distribution of losses with respect to par value')
ax.set_xlabel('Loss buckets')
plt.show()
nbsphinx-code-borderwhite
```

```
In [21]: cumsum = 0.
for i in range(len(loss_bucket_distribution)):
    cumsum += loss_bucket_distribution[i]
    print('{:24} | {:.06f}'.format(buckets[i], cumsum))
```

```
(      -0,      549000] | 0.469960
(   549000,    954000] | 0.689263
(   954000,    2097000] | 0.762897
(   2097000,    2502000] | 0.787553
(   2502000,    2835000] | 0.834408
(   2835000,    3168000] | 0.888119
(   3168000,    3573000] | 0.941345
(   3573000,    4716000] | 0.987143
(   4716000,    5121000] | 0.991483
(   5121000,    5670000] | 1.000000
```

## Impact of a belief of stress on a sector for the number of defaults in the portfolio

Belief from [1, 0] (no stress) to [1, 1] (no specific belief) to [0, 1] (stress in a sector).

```
In [22]: def show_expected_number_of_defaults(sector,q,enod_nobelief):
    v1=[]
    for i in range(101):
        posterior=gum.getPosterior(mn2, target='Number of defaults',evs={sector:[100-i,
↳i]})
        v1.append((posterior*q).sum())

    fig, ax = plt.subplots()
    ax.set_title(f'Expected number of defaults w.r.t stress in {sector}')
```

(continues on next page)



(continued from previous page)

```

ax.plot(np.arange(0,1.01,0.01),v1)
ax.plot(0.5,enod_nobelief,'x',color="red")
ax.text(0.5, enod_nobelief-0.05, ' No specific belief')
return fig

enod_nobelief=(gum.getPosterior(mn2, target='Number of defaults',evs={}) *q).sum()



gnb.flow.add(show_expected_number_of_defaults('Retail',q,enod_nobelief))
gnb.flow.add(show_expected_number_of_defaults('Finance',q,enod_nobelief))
gnb.flow.add(show_expected_number_of_defaults('Energy',q,enod_nobelief))
gnb.flow.display()

<IPython.core.display.HTML object>

```

In [ ]:

### 1.26.4 Learning and causality

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org">https://agrum.org</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>
--	---	---

```

In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

```

#### Model

Let's assume a process  $X_1 \rightarrow Y_1$  with a control on  $X_1$  by  $C_a$  and a parameter  $P_b$  on  $Y_1$ .

```

In [2]: bn=gum.fastBN("Ca->X1->Y1<-Pb")
bn.cpt("Ca").fillWith([0.8,0.2])
bn.cpt("Pb").fillWith([0.3,0.7])

bn.cpt("X1")[:]=[[0.9,0.1],[0.1,0.9]]

bn.cpt("Y1") [{"X1":0,"Pb":0}]=[0.8,0.2]
bn.cpt("Y1") [{"X1":1,"Pb":0}]=[0.2,0.8]
bn.cpt("Y1") [{"X1":0,"Pb":1}]=[0.6,0.4]
bn.cpt("Y1") [{"X1":1,"Pb":1}]=[0.4,0.6]

gnb.flow.row(bn,*[bn.cpt(x) for x in bn.nodes()])

<IPython.core.display.HTML object>

```

Actually the process is duplicated in the system but the control  $C_a$  and the parameter  $P_b$  are shared.

```

In [3]: bn.add("X2",2)
bn.add("Y2",2)

```

(continues on next page)

(continued from previous page)

```
bn.addArc("X2", "Y2")
bn.addArc("Ca", "X2")
bn.addArc("Pb", "Y2")

bn.cpt("X2").fillWith(bn.cpt("X1"), ["X1", "Ca"]) # copy cpt(X1) with the translation X2
↳ <-X1, Ca<-Ca
bn.cpt("Y2").fillWith(bn.cpt("Y1"), ["Y1", "X1", "Pb"]) # copy cpt(Y1) with translation
↳ Y2<-Y1, X2<-X1, Pb<-Pb

gnb.flow.row(bn, bn.cpt("X2"), bn.cpt("Y2"))

<IPython.core.display.HTML object>
```

## Simulation of the data

The process is partially observed : the control has been taken into account. However the parameter has not been identified and therefore is not collected.

```
In [4]: #the base will be saved in completeData="out/complete_data.csv", observedData="out/
↳ observed_data.csv"
import os
completeData="out/complete_data.csv"
observedData="out/observed_data.csv"
fixedObsData="res/fixed_observed_data.csv"

# generating complete date with pyAgrum
size=35000
#gum.generateSample(bn, 5000, "data.csv", random_order=True)
generator=gum.BNDatabaseGenerator(bn)
generator.setRandomVarOrder()
generator.drawSamples(size)
generator.toCSV(completeData)
```

```
In [5]: # selecting some variables using pandas
import pandas as pd
f=pd.read_csv(completeData)
keep_col = ["X1", "Y1", "X2", "Y2", "Ca"] # Pb is removed
new_f = f[keep_col]
new_f.to_csv(observedData, index=False)
```

In order to have fixed results, we will use now a database fixed\_observed\_data.csv generated with the process above.

## statistical learning

Using a classical statistical learning method, one can approximate a model from the observed data.

```
In [6]: learner=gum.BNlearner(fixedObsData)
learner.useGreedyHillClimbing()
bn2=learner.learnBN()
```

## Evaluating the impact of $X_2$ on $Y_1$

Using the database, a question for the user is to evaluate the impact of the value of  $X_2$  on  $Y_1$ .

```
In [7]: target="Y1"
      evs="X2"
      ie=gum.LazyPropagation(bn)
      ie2=gum.LazyPropagation(bn2)
      p1=ie.evidenceImpact(target,[evs])
      p2=gum.Potential(p1).fillWith(ie2.evidenceImpact(target,[evs]),[target,evs])
      errs=((p1-p2)/p1).scale(100)
      quaderr=(errs*errs).sum()
      gnb.flow.row(p1,p2,errs,f"${quaderr:3.5f}%",
                  captions=['in original model','in learned model','relative errors',
                             ↪'quadratic error'])
```

<IPython.core.display.HTML object>

## Evaluating the causal impact of $X_2$ on $Y_1$ with the learned model

The statistician notes that the change wanted by the user to apply on  $X_2$  is not an observation but rather an intervention.

```
In [8]: import pyAgrum.causal as csl
      import pyAgrum.causal.notebook as cslnb

      model=csl.CausalModel(bn)
      model2=csl.CausalModel(bn2)
```

```
In [9]: cslnb.showCausalModel(model)
      nbsphinx-code-borderwhite
```

```
In [10]: gum.config['notebook','graph_format']='svg'
      cslnb.showCausalImpact(model,target,{evs})
      cslnb.showCausalImpact(model2,target,{evs})
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Unfortunately, due to the fact that  $P_a$  is not learned, the computation of the causal impact still is imprecise.

```
In [11]: _, impact1, _ = csl.causalImpact(model, on=target, doing={evs})
      _, impact2orig, _ = csl.causalImpact(model2, on=target, doing={evs})

      impact2=gum.Potential(p2).fillWith(impact2orig,['Y1','X2'])
      errs=((impact1-impact2)/impact1).scale(100)
      quaderr=(errs*errs).sum()
      gnb.flow.row(impact1,impact2,errs,f"${quaderr:3.5f}%",
                  captions=['$P( Y_1 \mid \hookrightarrow X_2)$ <br/>in original model',
                             '$P( Y_1 \mid \hookrightarrow X_2)$ <br/>in learned model',
                             ↪<br/>relative errors',' <br/>quadratic error'])
```

<IPython.core.display.HTML object>

Just to be certain, we can verify that in the original model,  $P(Y_1 \mid \hookrightarrow X_2) = P(Y_1)$

```
<IPython.core.display.HTML object>
```

```
learner.setUserC()  
bn3=learner.learnBN()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```



There are several sensitivity analysis frameworks for Bayesian networks. A fairly efficient method is certainly to use credal networks to do this analysis.

### Creating a Bayesian network

```
In [1]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

```
In [2]: bn=gum.fastBN("A->B->C<-D->E->F<-B")
gnb.flow.row(bn,gnb.getInference(bn))

<IPython.core.display.HTML object>
```

### Building a credal network from a BN

It is easy to build a credal network from a Bayesian network by indicating the ‘noise’ on each parameter.

```
In [3]: cr=gum.CredalNet(bn,bn)
gnb.show(cr)
nbsphinx-code-borderwhite
```

```
In [4]: cr.bnToCredal(beta=1e-10,oneNet=False)
cr.computeBinaryCPTMinMax()
print(cr)
```

```
A:Range([0,1])
<> : [[0.182055 , 0.817945] , [0.132709 , 0.867291]]

B:Range([0,1])
<A:0> : [[0.814242 , 0.185758] , [0.814241 , 0.185759]]
<A:1> : [[0.360802 , 0.639198] , [0.359964 , 0.640036]]

C:Range([0,1])
<B:0|D:0> : [[0.187328 , 0.812672] , [0.15419 , 0.84581]]
<B:1|D:0> : [[0.507362 , 0.492638] , [0.507283 , 0.492717]]
<B:0|D:1> : [[0.425391 , 0.574609] , [0.425104 , 0.574896]]
<B:1|D:1> : [[0.184355 , 0.815645] , [0.112276 , 0.887724]]

D:Range([0,1])
<> : [[0.984151 , 0.0158492]]

E:Range([0,1])
<D:0> : [[0.18389 , 0.81611] , [0.143465 , 0.856535]]
<D:1> : [[0.635584 , 0.364416] , [0.635572 , 0.364428]]

F:Range([0,1])
<E:0|B:0> : [[0.688333 , 0.311667] , [0.688327 , 0.311673]]
<E:1|B:0> : [[0.242911 , 0.757089] , [0.235478 , 0.764522]]
<E:0|B:1> : [[0.350608 , 0.649392] , [0.34961 , 0.65039]]
<E:1|B:1> : [[0.529391 , 0.470609] , [0.529335 , 0.470665]]
```

## Testing difference hypothesis about the global precision on the parameters

We can therefore easily conduct a sensitivity analysis based on an assumption of error on all the parameters of the network.



```
In [5]: def showNoisy(bn,beta):
        cr=gum.CredalNet(bn,bn)
        cr.bnToCredal(beta=beta,oneNet=False)
        cr.computeBinaryCPTMinMax()
        ielbp=gum.CNLoopyPropagation(cr)
        return gnb.getInference(cr,engine=ielbp)

In [6]: for eps in [1,1e-3,1e-5,1e-8,1e-10]:
        gnb.flow.add(showNoisy(bn,eps),caption=f"noise={eps}")
        gnb.flow.display()

<IPython.core.display.HTML object>
```

```
In [ ]:
```

### 1.26.6 Quasi-continuous BN

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org"> (https://agrums.org) </a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html) </a>
--	---	---

```
In [1]: from pylab import *
import matplotlib.pyplot as plt
```

aGrUM cannot (currently) deal with continuous variables. However, a discrete variable with a large enough domain size is an approximation of such variables.

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

#nbr of states for quasi continuous variables. You can change the value
#but be careful of the quadratic behavior of both memory and time complexity
#in this example.
minB,maxB=-3,3
minC,maxC=4,14
NB=300
```

```
In [3]: bn=gum.BayesNet("Quasi-Continuous")
bn.add(gum.LabelizedVariable("A","A binary variable",2))
bn.add(gum.NumericalDiscreteVariable("B","A range variable",minB,maxB,NB))
bn.addArc("A","B")
print(bn)
gnb.showBN(bn)
```

```
BN{nodes: 2, arcs: 1, domainSize: 600, dim: 602}
nbsphinx-code-borderwhite
```

```
In [4]: bn.cpt("A")[:]=[0.4, 0.6]
gnb.showProba(bn.cpt("A"))
nbsphinx-code-borderwhite
```

### CPT for quasi-continuous variables (with parents)

Using python (and scipy), it is easy to find pdf for continuous variable

```
In [5]: # we truncate a pdf, so we need to normalize
def normalize(rv,vmin,vmax,size):
    pdf=rv.pdf(linspace(vmin,vmax,size))
    return (pdf/sum(pdf))

from scipy.stats import norm,genhyperbolic
p, a, b = 0.5, 1.5, -0.7
bn.cpt("B")[:,0]=normalize(norm(2.41),minB,maxB,NB)
bn.cpt("B")[:,1]=normalize(genhyperbolic(p,a,b),minB,maxB,NB)
gnb.flow.clear()
gnb.flow.add(gnb.getProba(bn.cpt("B").extract({"A":0})),caption="P(B|A=0)")
gnb.flow.add(gnb.getProba(bn.cpt("B").extract({"A":1})),caption="P(B|A=1)")
gnb.flow.display()

<IPython.core.display.HTML object>
```

### Quasi-continuous inference (with no evidence)

```
In [6]: gnb.showPosterior(bn,target="B",evs={})
nbsphinx-code-borderwhite
```

### Quasi-continuous variable with quasi-continuous parent

```
In [7]: bn.add(gum.NumericalDiscreteVariable("C","Another quasi continuous variable",minC,
    ↪maxC,NB))
bn.addArc("B","C")
gnb.showBN(bn) # B and C are quasi-continouous
nbsphinx-code-borderwhite
```

Even if this BN is quite small (and linear), the size of nodes  $B$  et  $C$  are rather big and creates a complex model ( $NB \times NB$  parameters in  $P(C|B)$ ).

```
In [8]: print("nombre de paramètres du bn : {}".format(bn.dim()))
print("domaine du bn : 10^{0}".format(bn.log10DomainSize()))

nombre de paramètres du bn : 90299
domaine du bn : 10^5.2552725051033065
```

```
In [9]: help(gnb.flow.add)

Help on method add in module pyAgrum.lib.notebook:

add(obj, caption=None, title=None) method of pyAgrum.lib.notebook.FlowLayout instance
    add an element in the row by trying to treat it as plot or html if possible.
```

(continues on next page)

(continued from previous page)

`(title is an obsolete parameter)`

```
In [10]: from scipy.stats import gamma
# cpt("C") is NB x NB matrix !
l=[]
for i in range(NB):
    k=(i*10.0)/NB
    l.append(normalize(gamma(k+1),4,14,NB))

bn.cpt("C")[:,]=l

def showB(n:int):
    gnb.flow.add(gnb.getProba(bn.cpt("C").extract({"B":n})),
                 caption=f"P(C|B={bn.variable('B').label(n)})")

gnb.flow.clear()
showB(0)
showB(NB//4)
showB(NB*2//3)
showB(NB-1)
gnb.flow.display()

<IPython.core.display.HTML object>
```

## Inference in quasi-continuous BN

```
In [11]: import time

ts = time.time()
ie=gum.LazyPropagation(bn)
ie.makeInference()
q=ie.posterior("C")
te=time.time()
gnb.flow.add(gnb.getPosterior(bn,target="C",evs={}),caption=f"P(C) computed in {te-ts:
↪2.5f} sec for a model with {bn.dim()} paramters")
gnb.flow.display()

<IPython.core.display.HTML object>
```

## Changing prior

```
In [12]: bn.cpt("A")[:,]=[0.9,0.1]

gnb.flow.add(gnb.getPosterior(bn,target="C",evs={}),caption="P(C) with P(A)=[0.9,0.1]
↪")
gnb.flow.display()

<IPython.core.display.HTML object>
```



## inference with evidence in quasi-continuous BN

We want to compute

$$P(A|C = 9)$$

$$P(B|C = 9)$$

```
In [13]: ie=gum.LazyPropagation(bn)
         ie.setEvidence({'C':bn.variable("C").toNumericalDiscreteVar().closestLabel(9)})
         ie.makeInference()
         plot(linspace(minB,maxB,NB),ie.posterior("B")[:])
         title("P( B | C={0} )".format(bn.variable("C").toNumericalDiscreteVar().
         ↪closestLabel(9)));
         nbsphinx-code-borderwhite
```

```
In [14]: gnb.showPosterior(bn,target="B",evs={'C':bn.variable("C").toNumericalDiscreteVar().
         ↪closestLabel(9)})
         nbsphinx-code-borderwhite
```

```
In [15]: gnb.showProba(ie.posterior("A"))
         nbsphinx-code-borderwhite
```

## Multiple inference : MAP DECISION between Gaussian and generalized hyperbolic distributions

What is the behaviour of  $P(A|C = i)$  when  $i$  varies ? I.e. we perform a MAP decision between the two models ( $A = 0$  for the Gaussian distribution and  $A = 1$  for the generalized hyperbolic distribution).

```
In [16]: bn.cpt("A")[:] = [0.1, 0.9]
         ie=gum.LazyPropagation(bn)
         p0=[]
         p1=[]
         for i in bn.variable("C").labels():
             ie.setEvidence({'C':i})
             ie.makeInference()
             p0.append(ie.posterior("A")[0])
             p1.append(ie.posterior("A")[1])

         x=[float(v) for v in bn.variable("C").labels()]
         plot(x,p0)
         plot(x,p1)
         title("P( A | C=i ) with prior p(A)=[0.1,0.9]")
         legend(["A=0", "A=1"],loc='best')
         inters=(transpose(p0)<transpose(p1)).argmin()

         text(x[inters]-0.2,p0[inters],
              "{0},{1:5.4f} ".format(x[inters],p0[inters]),
              bbox=dict(facecolor='red', alpha=0.1),ha='right');
         nbsphinx-code-borderwhite
```

i.e. if  $C < 13.2308$  then  $A = 1$  else  $A = 0$

Changing the prior  $P(A)$ 

```
In [17]: bn.cpt("A").fillWith([0.4, 0.6])
ie=gum.LazyPropagation(bn)
p0=[]
p1=[]
for i in range(300):
    ie.setEvidence({'C':i})
    ie.makeInference()
    p0.append(ie.posterior("A")[0])
    p1.append(ie.posterior("A")[1])
x=[float(v) for v in bn.variable("C").labels()]
plot(x,p0)
plot(x,p1)
title("P( A | C=i) with prior p(A)=[0.1,0.9]")
legend(["A=0", "A=1"],loc='best')
inters=(transpose(p0)<transpose(p1)).argmin()



text(x[inters]+0.2,p0[inters],
     "{0},{1:5.4f}".format(x[inters],p0[inters]),
     bbox=dict(facecolor='red', alpha=0.1),ha='left');
nbsphinx-code-borderwhite
```

ie. with  $p(A) = [0.4, 0.6]$ , if  $C < 7.8462$  then  $A = 1$  else  $A = 0$ .

In [ ]:

## 1.26.7 Parameter learning with Pandas

This notebook uses pandas to learn the parameters. However **the simplest way to learn parameters is to use ``BNLearner``** :-). Moreover, you will be able to add priors, etc (see learning BN).

 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrum.org">https://agrum.org</a>)</p>	<p>(<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html">https://agrum.gitlab.io/extra/agrum_at_binder.html</a>)</p>
--	--	--

```
In [1]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt

import os
```

## Importing pyAgrum

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

## Loading two BNs

```
In [3]: bn=gum.loadBN("res/asia.bif")
bn2=gum.loadBN("res/asia.bif")

gnb.sideBySide(bn,bn2,
               captions=['First bn','Second bn'])

<IPython.core.display.HTML object>
```

## Randomizing the parameters

```
In [4]: bn.generateCPTs()
bn2.generateCPTs()
```

## Direct comparison of parameters

```
In [5]: from IPython.display import HTML

gnb.sideBySide(bn.cpt(3),
               bn2.cpt(3),
               captions=['<h3>cpt of node 3 in first bn</h3>','<h3>same cpt in second_
↳bn</h3>'])

<IPython.core.display.HTML object>
```

## Exact KL-divergence

Since the BN is not too big, BruteForceKL can be computed ...

```
In [6]: g1=gum.ExactBNdistance(bn,bn2)
before_learning=g1.compute()
print(before_learning['klPQ'])

5.0030971388003485
```

Just to be sure that the distance between a BN and itself is 0 :

```
In [7]: g0=gum.ExactBNdistance(bn,bn)
print(g0.compute()['klPQ'])

0.0
```

## Generate a database from the original BN

```
In [8]: gum.generateSample(bn,10000,"out/test.csv",True)
```

```
out/test.csv: 100%| |
```

```
Log2-Likelihood : -62895.748932625655
```

```
Out[8]: -62895.748932625655
```

## Using pandas for \_counting

As an exercise, we will use pandas to learn the parameters.

```
In [9]: # using bn as a template for the specification of variables in test.csv
```

```
learner=gum.BNlearner("out/test.csv",bn)
```

```
bn3=learner.learnParameters(bn.dag())
```

```
#the same but we add a Laplace adjustment (smoothing) as a Prior
```

```
learner=gum.BNlearner("out/test.csv",bn)
```

```
learner.useSmoothingPrior(1000) # a count C is replaced by C+1000
```

```
bn4=learner.learnParameters(bn.dag())
```

```
after_pyAgrum_learning=gum.ExactBNdistance(bn,bn3).compute()
```

```
after_pyAgrum_learning_with_smoothing=gum.ExactBNdistance(bn,bn4).compute()
```

```
print("without prior:{}".format(after_pyAgrum_learning['klPQ']))
```

```
print("with prior smooting(1000):{}".format(after_pyAgrum_learning_with_smoothing[
↪ 'klPQ']))
```

```
without prior:0.0018130896108582272
```

```
with prior smooting(1000):0.23166819139971712
```

## Now, let's try to learn the parameters with pandas

```
In [10]: import pandas
```

```
In [11]: # We directly generate samples in a DataFrame
```

```
df,_=gum.generateSample(bn,10000,None,True)
```

```
100%| |
```

```
Log2-Likelihood : -62901.86568237794
```

```
In [12]: df.head()
```

```
Out[12]:   bronchitis  smoking  tuberculosis  visit_to_Asia  positive_XraY  lung_cancer  \
0           0         1             0                0              1           0
1           1         1             1                1              1           1
2           1         1             1                1              0           0
3           1         1             1                1              1           0
4           1         1             0                1              0           0

   tuberculos_or_cancer  dyspnoea
0                     1         0
```

(continues on next page)

(continued from previous page)

1	1	1
2	0	1
3	1	0
4	1	1

We use the crosstab function in pandas

```
In [13]: c=pandas.crosstab(df['dyspnoea'],[df['tuberculos_or_cancer'],df['bronchitis']])
c
```

```
Out[13]: tuberculos_or_cancer    0      1
bronchitis      0      1      0      1
dyspnoea
0              563  1484  2348  1889
1             1000   247   927  1542
```

Playing with numpy reshaping, we retrieve the good form for the CPT from the pandas cross-table

```
In [14]: gnb.sideBySide('<pre>'+str(np.array((c/c.sum()).apply(np.float32)).transpose()).
↳reshape(2,2,2))+</pre>',
          bn.cpt(bn.idFromName('dyspnoea')),
          captions=["<h3>Learned parameters in crosstab", "<h3>Original_
↳parameters in bn</h3>"])
<IPython.core.display.HTML object>
```

## A global method for estimating Bayesian network parameters from CSV file using PANDAS

```
In [15]: def computeCPTfromDF(bn,df,name):
        """
        Compute the CPT of variable "name" in the BN bn from the database df
        """
        id=bn.idFromName(name)
        parents=list(reversed(bn.cpt(id).names))
        domains=[bn.variableFromName(name).domainSize()
                  for name in parents]
        parents.pop()

        if (len(parents)>0):
            c=pandas.crosstab(df[name],[df[parent] for parent in parents])
            s=c/c.sum().apply(np.float32)
        else:
            s=df[name].value_counts(normalize=True)

        bn.cpt(id)[:]=np.array((s).transpose()).reshape(*domains)

def ParametersLearning(bn,df):
    """
    Compute the CPTs of every variable in the BN bn from the database df
    """
    for name in bn.names():
        computeCPTfromDF(bn,df,name)
```

```
In [16]: ParametersLearning(bn2,df)
```

KL has decreased a lot (if everything's OK)

```
In [17]: g1=gum.ExactBNdistance(bn,bn2)
print("BEFORE LEARNING")
print(before_learning['klPQ'])
print
print("AFTER LEARNING")
print(g1.compute()['klPQ'])
```

```
BEFORE LEARNING
5.0030971388003485
AFTER LEARNING
2.7808990516850507
```

And CPTs should be close

```
In [18]: gnb.sideBySide(bn.cpt(3),
                      bn2.cpt(3),
                      captions=["<h3>Original BN", "<h3>learned BN</h3>"])

<IPython.core.display.HTML object>
```

### Influence of the size of the database on the quality of learned parameters

What is the effect of increasing the size of the database on the KL ? We expect that the KL decreases to 0.

```
In [19]: res=[]
for i in range(200,10001,50):
    ParametersLearning(bn2,df[:i])
    g1=gum.ExactBNdistance(bn,bn2)
    res.append(g1.compute()['klPQ'])
fig=figure(figsize=(10,6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(range(200,10001,50),res)
ax.set_xlabel("size of the database")
ax.set_ylabel("KL")
ax.set_title("klPQ(bn,learnedBN(x))");
nbsphinx-code-borderwhite
```

In [ ]:

### 1.26.8 Bayesian Beta Distributed Coin Inference



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

## build a fully bayesian beta distributed coin inference

This notebook is based on examples from Benjamin Datko (<https://gist.github.com/bdatko>).

The basic idea of this notebook is to show you could assess the probability for a coin, knowing a sequence of heads/tails.

```
In [1]: import itertools
import time

from pylab import *
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

```
In [2]: gum.config["notebook","default_graph_size"]="12!"
gum.config["notebook","default_graph_inference_size"]="12!"
```

### 1.26.9 Fill Beta parameters with a re-parameterization

- [https://en.wikipedia.org/wiki/Beta\\_distribution](https://en.wikipedia.org/wiki/Beta_distribution)

Alternative parameterizations [\[ edit \]](#)

Two parameters [\[ edit \]](#)

Mean and sample size [\[ edit \]](#)

The beta distribution may also be reparameterized in terms of its mean  $\mu$  ( $0 < \mu < 1$ ) and the sum of the two shape parameters  $\nu = \alpha + \beta > 0$  ([\[9\]](#) p. 83). Denoting by  $\alpha$ Posterior and  $\beta$ Posterior the shape parameters of the posterior beta distribution resulting from applying Bayes theorem to a binomial likelihood function and a prior probability, the interpretation of the addition of both shape parameters to be sample size =  $\nu = \alpha$ Posterior +  $\beta$ Posterior is only correct for the Haldane prior probability Beta(0,0). Specifically, for the Bayes (uniform) prior Beta(1,1) the correct interpretation would be sample size =  $\alpha$ Posterior +  $\beta$ Posterior - 2, or  $\nu$  = (sample size) + 2. For sample size much larger than 2, the difference between these two priors becomes negligible. (See section [Bayesian inference](#) for further details.)  $\nu = \alpha + \beta$  is referred to as the "sample size" of a Beta distribution, but one should remember that it is, strictly speaking, the "sample size" of a binomial likelihood function only when using a Haldane Beta(0,0) prior in Bayes theorem.

This parameterization may be useful in Bayesian parameter estimation. For example, one may administer a test to a number of individuals. If it is assumed that each person's score ( $0 \leq \theta \leq 1$ ) is drawn from a population-level Beta distribution, then an important statistic is the mean of this population-level distribution. The mean and sample size parameters are related to the shape parameters  $\alpha$  and  $\beta$  via<sup>[\[9\]](#)</sup>

$$\alpha = \mu\nu, \beta = (1 - \mu)\nu$$

Under this [parameterization](#), one may place an [uninformative prior](#) probability over the mean, and a vague prior probability (such as an exponential or gamma distribution) over the positive reals for the sample size, if they are independent, and prior data and/or beliefs justify it.

We propose a model where : mu and nu are the parameters of a beta which gives the distribution for the coins.

- below are some useful definitions

$$\alpha = \mu\nu$$

$$\beta = (1 - \mu)\nu$$

$$\mu = \frac{\alpha}{\alpha + \beta}$$

- like in Wikipedia article, we will have a uniform prior on  $\mu$  and an exponential prior on  $\nu$

```
In [3]: # the sequence of COINS
serie=[1,0,0,0,1,0,1,1,0,1,0,0,1,0,0,1]
```

```
In [4]: def fillvect(rv,vmin,vmax,size, **pdf_kwargs):
    x = linspace(vmin,vmax,size)
    pdf=rv.pdf(x, **pdf_kwargs)
    return x,pdf

def normalize(rv,vmin,vmax,size, **pdf_kwargs):
    x = linspace(vmin,vmax,size)
    pdf=rv.pdf(x, **pdf_kwargs)
    return x,(pdf/sum(pdf))
```

```
In [5]: NB_ = 300
        vmin, vmax = 0.001, 0.999
        pmin_mu, pmax_mu = 0.001, 0.999
        pmin_nu, pmax_nu = 1, 50
        size_ = 16
```

```
In [6]: bn=gum.BayesNet("SEQUENCE OF COINS MODEL")
        mu = bn.add(gum.NumericalDiscreteVariable("mu","mean of the Beta distribution",0,1,NB_
        ↪))
        nu = bn.add(gum.NumericalDiscreteVariable("nu","'sample size' of the Beta where nu =_
        ↪a + b > 0",0,50,NB_))
        bias=bn.add(gum.NumericalDiscreteVariable("bias","The bias of the coin",0,1,NB_))
        hs=[bn.add(gum.LabelizedVariable(f"H{i}", "The hallucinations of coin flips",2)) for i_
        ↪in range(size_)]

        bn.addArc(mu,bias)
        bn.addArc(nu,bias)
        for h in hs:
            bn.addArc(bias,h)
        print(bn)
        bn
```

```
BN{nodes: 19, arcs: 18, domainSize: 10^12.2478, dim: 27010200}
```

```
Out[6]: (pyAgrum.BayesNet<double>@0000017F2FEDD2A0) BN{nodes: 19, arcs: 18, domainSize: 10^12.
        ↪2478, dim: 27010200}
```

```
In [7]: loc_, scale_ = 2, 5
        x_nu, y_nu = normalize_(scipy.stats.expon,pmin_nu,pmax_nu,NB_,loc=loc_, scale=scale_)
        x_mu, y_mu = normalize_(scipy.stats.uniform,pmin_mu,pmax_mu,NB_,)

        bn.cpt(mu)[:]= y_mu # uniform prior for hyperparameter
        bn.cpt(nu)[:]= y_nu # expoential prior for hyperparameter

        gnb.flow.clear()
        gnb.flow.add(gnb.getProba(bn.cpt(nu)),caption="Distribution for nu")
        gnb.flow.add(gnb.getProba(bn.cpt(mu)),caption="Distribution for mu")
        gnb.flow.display()
```

```
<IPython.core.display.HTML object>
```

```
In [8]: # https://scicomp.stackexchange.com/a/10800
        al_ = (x_mu[:,newaxis] * x_nu[newaxis,:])
        be_ = (1 - x_mu)[:,newaxis] * x_nu[newaxis,:]

        t_start = time.time()
        pdf = scipy.stats.beta(al_,be_).pdf(linspace(vmin, vmax,NB_)[: ,newaxis, newaxis])
        bn.cpt("bias").fillWith(np.swapaxes(pdf, 0, -1).flatten())
        bn.cpt("bias").normalizeAsCPT()
        end_time = time.time() - t_start
        print(f"Filling {NB_}^3 parameters in {end_time:5.3f}s")
```

```
Filling 300^3 parameters in 7.695s
```

```
In [9]: x_bias = linspace(vmin, vmax, NB_)
        x_hs = np.array([1 - x_bias, x_bias]).T
        for h in hs:
            bn.cpt(h).fillWith(x_hs.flatten()).normalizeAsCPT()
```



## Evidence with the sequence

In [ ]:

### 1.27.1 Potentials : named tensors

 <p>(<a href="http://creativecommons.org/licenses/by-nc/4.0/">http://creativecommons.org/licenses/by-nc/4.0/</a>)</p>	 <p>(<a href="https://agrum.org">https://agrum.org</a>)</p>	<p>(<a href="https://agrum.gitlab.io/extra/agrum">https://agrum.gitlab.io/extra/agrum</a>)</p>
--	--	--

(continues on next page)

(continued from previous page)

```
a,b,c=[gum.LabelizedVariable(s,s,2) for s in "abc"]
```

### potential algebra

```
In [2]: p1=gum.Potential().add(a).add(b).fillWith([1,2,3,4]).normalize()
p2=gum.Potential().add(b).add(c).fillWith([4,5,2,3]).normalize()
```

```
In [3]: gnb.flow.row(p1,p2,p1+p2,
                  captions=['p1', 'p2', 'p1+p2'])

<IPython.core.display.HTML object>
```

```
In [4]: p3=p1+p2
gnb.showPotential(p3/p3.margSumOut(["b"]))

<IPython.core.display.HTML object>
```

```
In [5]: p4=gum.Potential()+p3
gnb.flow.row(p3,p4,
             captions=['p3', 'p4'])

<IPython.core.display.HTML object>
```

### Bayes' theorem

```
In [6]: bn=gum.fastBN("a->c;b->c",3)
bn
```

```
Out[6]: (pyAgrum.BayesNet<double>@00000029D371A81A0) BN{nodes: 3, arcs: 2, domainSize: 27, dim:
↪ 33}
```

In such a small bayes net, we can directly manipulate  $P(a, b, c)$ . For instance :

$$P(b|c) = \frac{\sum_a P(a, b, c)}{\sum_{a,b} P(a, b, c)}$$

```
In [7]: pABC=bn.cpt("a")*bn.cpt("b")*bn.cpt("c")
pBgivenC=(pABC.margSumOut(["a"])/pABC.margSumOut(["a", "b"]))

pBgivenC.putFirst("b") # in order to have b horizontally in the table
```

```
Out[7]: (pyAgrum.Potential<double>@00000029D371494D0)
```

	b		
c	0	1	2
0	0.3267	0.1174	0.5559
1	0.4532	0.1154	0.4314
2	0.4143	0.2231	0.3626

**Joint, marginal probability, likelihood**

Let's compute the joint probability  $P(A, B)$  from  $P(A, B, C)$

```
In [8]: pAC=pABC.margSumOut(["b"])
print("pAC really is a probability : it sums to {}".format(pAC.sum()))
pAC
```

```
pAC really is a probability : it sums to 1.0000000000000002
```

```
Out[8]: (pyAgrum.Potential<double>@00000029D371497B0)
```

	a		
c	0	1	2
0	0.0156	0.0621	0.2078
1	0.0206	0.1758	0.1153
2	0.0337	0.1927	0.1765

**Computing  $p(A)$** 

```
In [9]: pAC.margSumOut(["c"])
```

```
Out[9]: (pyAgrum.Potential<double>@00000029D37149610)
```

a			
0	1	2	
0.0699	0.4306	0.4996	

**Computing  $p(A|C = 1)$** 

It is easy to compute  $p(A, C = 1)$

```
In [10]: pAC.extract({"c":1})
```

```
Out[10]: (pyAgrum.Potential<double>@00000029D37149830)
```

a			
0	1	2	
0.0206	0.1758	0.1153	

Moreover, we know that  $P(C = 1) = \sum_A P(A, C = 1)$

```
In [11]: pAC.extract({"c":1}).sum()
```

```
Out[11]: 0.3116561031542595
```

Now we can compute  $p(A|C = 1) = \frac{P(A, C=1)}{p(C=1)}$

```
In [12]: pAC.extract({"c":1}).normalize()
```

```
Out[12]: (pyAgrum.Potential<double>@00000029D37149570)
```

a			
0	1	2	
0.0660	0.5640	0.3700	

**Computing  $P(A|C)$** 

$P(A|C)$  is represented by a matrix that verifies  $p(A|C) = \frac{P(A,C)}{P(C)}$

```
In [13]: pAgivenC=(pAC/pAC.margSumIn("c")).putFirst("a")
# putFirst("a") : to correctly show a cpt, the first variable have to be the
↳ conditioned one
gnb.flow.row(pAgivenC,pAgivenC.extract({'c':1}),
             captions=["$P(A|C)$", "$P(A|C=1)$"])

<IPython.core.display.HTML object>
```

**Likelihood  $P(A = 2|C)$** 

A likelihood can also be found in this matrix.

```
In [14]: pAgivenC.extract({'a':2})
Out[14]: (pyAgrum.Potential<double>@00000029D37149630)
      c
      |
0      | 1      | 2      |
-----|-----|-----|
0.7278 | 0.3700 | 0.4381 |
```

A likelihood does not have to sum to 1. It is not relevant to normalize it.

```
In [15]: pAgivenC.margSumIn(["a"])
Out[15]: (pyAgrum.Potential<double>@00000029D371497F0)
      a
      |
0      | 1      | 2      |
-----|-----|-----|
0.2043 | 1.2598 | 1.5359 |
```

**entropy of potential**

```
In [16]: %matplotlib inline
from pylab import *
import matplotlib.pyplot as plt
import numpy as np
```

```
In [17]: p1=gum.Potential().add(a)
x = np.linspace(0, 1, 100)
plt.plot(x,[p1.fillWith([p,1-p]).entropy() for p in x])
plt.show()
nbsphinx-code-borderwhite
```

```
In [18]: t=gum.LabelizedVariable('t','t',3)
p1=gum.Potential().add(t)

def entrop(bc):
    """
    bc is a list [a,b,c] close to a distribution
    (normalized just to be sure)
    """
    return p1.fillWith(bc).normalize().entropy()
```

(continues on next page)



(continued from previous page)

```
import pyAgrum.lib.notebook as gnb
```

```
In [2]: min_x=0
max_x=15

bn=gum.BayesNet()
l=[bn.add(gum.RangeVariable(item,item,min_x,max_x)) for item in ["a","b","c","d","e",
↪ "f"]]

gum.config['notebook','histogram_line_threshold']=15
```

```
In [3]: nmax=bn.addMAX(gum.RangeVariable("MAX","MAX",min_x,max_x))
bn.addArc(l[0],nmax)
bn.addArc(l[1],nmax)
bn.addArc(l[2],nmax)

nmin=bn.addMIN(gum.RangeVariable("MIN","MIN",min_x,max_x))
bn.addArc(l[3],nmin)
bn.addArc(l[4],nmin)
bn.addArc(l[5],nmin)

nampl=bn.addAMPLITUDE(gum.RangeVariable("DELTA","DELTA",0,max_x-min_x))
bn.addArc(nmax,nampl)
bn.addArc(nmin,nampl)

nmedian=bn.addMEDIAN(gum.RangeVariable("MEDIAN","MEDIAN",min_x,max_x))
for n in [l[0],l[1],l[2],l[3]]:
    bn.addArc(n,nmedian)
#potential for median has a size : 16^5=2^20 double !

nexists=bn.addEXISTS(gum.LabelizedVariable("EXISTS_0","EXISTS"),0)
bn.addArc(l[0],nexists)
bn.addArc(l[1],nexists)
bn.addArc(l[2],nexists)

nforall=bn.addFORALL(gum.LabelizedVariable("FORALL_1","FORALL"),1)
bn.addArc(l[3],nforall)
bn.addArc(l[4],nforall)
bn.addArc(l[5],nforall)

ncount=bn.addCOUNT(gum.RangeVariable("COUNT_1","COUNT_1",0,3),1)
bn.addArc(l[0],ncount)
bn.addArc(l[1],ncount)
bn.addArc(l[2],ncount)
```

```
In [4]: for nod in l:
        bn.cpt(nod).fillWith(1).normalize()
gnb.showInference(bn,size="13")
nbsphinx-code-borderwhite
```

```
In [5]: gnb.showInference(bn,size="13",evs={'MEDIAN':[0,0,0,0,0,0,0,1,1,1,1,1,0,0]})
nbsphinx-code-borderwhite
```

```
In [6]: # if the roots do not have uniform but random distribution
```

(continues on next page)

(continued from previous page)

```
for nod in l:
    bn.generateCPT(nod)

gnb.showInference(bn,size="13")
nbsphinx-code-borderwhite
```

```
In [7]: gnb.showInference(bn,size="13",evs={'MEDIAN':[0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0]})
nbsphinx-code-borderwhite
```

```
In [ ]:
```

### 1.27.3 Explaining a model



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrums.org>)

([https://agrums.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrums.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: import time

from pyAgrum.lib.bn2graph import BN2dot
import numpy as np
import pandas as pd

import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.explain as expl

import matplotlib.pyplot as plt
```

#### Building the model

We build a simple graph for the example

```
In [2]: template=gum.fastBN("X1->X2->Y;X3->Z->Y;X0->Z;X1->Z;X2->R[5];Z->R;X1->Y")
data_path = "res/shap/Data_6var_direct_indirect.csv"

#gum.generateSample(template,1000,data_path)

learner = gum.BNlearner(data_path,template)
bn = learner.learnParameters(template.dag())
bn
```

```
Out[2]: (pyAgrum.BayesNet<double>@000000236AF7DC340) BN{nodes: 7, arcs: 9, domainSize: 320,
↳ dim: 62}
```

## 1-independence list (w.r.t. the class Y)

Given a model, it may be interesting to investigate the conditional independences of the class Y created by this very model.

```
In [3]: # this function explores all the CI between 2 variables and computes the p-values w.r.
        ↪t to a csv file.
```

```
expl.independenceListForPairs(bn,data_path)
```

```
Out[3]: {('R', 'X0', ('X1', 'Z')): 0.7083382647903902,
        ('R', 'X1', ('X2', 'Z')): 0.46938486254099493,
        ('R', 'X3', ('X1', 'Z')): 0.4128522974536623,
        ('R', 'Y', ('X2', 'Z')): 0.8684231094674686,
        ('X0', 'X1', ()): 0.723302358657366,
        ('X0', 'X2', ()): 0.9801394906304377,
        ('X0', 'X3', ()): 0.7676868597218647,
        ('X0', 'Y', ('X1', 'Z')): 0.5816487109659612,
        ('X1', 'X3', ()): 0.5216508257424717,
        ('X2', 'X3', ()): 0.9837021981131505,
        ('X2', 'Z', ('X1', 'Z')): 0.6638491605436834,
        ('X3', 'Y', ('X1', 'Z')): 0.8774081450472304}
```

nbsphinx-code-borderwhite

... with respect to a specific target.

```
In [4]: expl.independenceListForPairs(bn,data_path,target="Y")
```

```
Out[4]: {('Y', 'R', ('X2', 'Z')): 0.8684231094674686,
        ('Y', 'X0', ('X1', 'Z')): 0.5816487109659612,
        ('Y', 'X3', ('X1', 'Z')): 0.8774081450472304}
```

nbsphinx-code-borderwhite

## 2-ShapValues

```
In [5]: print(expl.ShapValues.__doc__)
```

The ShapValue class implements the calculation of Shap values in Bayesian networks.

The main implementation is based on Conditional Shap values [3]\_, but the ↪  
↪Interventional calculation method proposed in [2]\_ is also present. In addition, a ↪  
↪new causal method, based on [1]\_, is implemented which is well suited for Bayesian ↪  
↪networks.

.. [1] Heskens, T., Sijben, E., Bucur, I., & Claassen, T. (2020). Causal Shapley ↪  
↪Values: Exploiting Causal Knowledge. 34th Conference on Neural Information ↪  
↪Processing Systems. Vancouver, Canada.

.. [2] Janzing, D., Minorics, L., & Blöbaum, P. (2019). Feature relevance ↪  
↪quantification in explainable AI: A causality problem. arXiv: Machine Learning. ↪  
↪Retrieved 6 24, 2021, from <https://arxiv.org/abs/1910.13413>

.. [3] Lundberg, S. M., & Su-In, L. (2017). A Unified Approach to Interpreting Model. ↪  
↪31st Conference on Neural Information Processing Systems. Long Beach, CA, USA.

The ShapValue class implements the calculation of Shap values in Bayesian networks. It is necessary to specify a target and to provide a Bayesian network whose parameters are known and will be used later in the different calculation methods.



```
In [6]: gumshap = expl.ShapValues(bn, 'Y')
```

### Compute Conditionnal in Bayesian Network

A dataset (as a `pandas.dataframe`) must be provided so that the Bayesian network can learn its parameters and then predict.

The method `conditional` computes the conditonal shap values using the Bayesian Networks. It returns 2 graphs and a dictionary. The first one shows the distribution of the shap values for each of the variables, the second one classifies the variables by their importance.

```
In [7]: train = pd.read_csv(data_path).sample(frac=1.)
```

```
In [8]: t_start = time.time()
resultat = gumshap.conditional(train, plot=True,plot_importance=True,percentage=False)
print(f'Run Time : {time.time()-t_start} sec')
```

```
Run Time : 6.830995082855225 sec
```

```
nbsphinx-code-borderwhite
```

```
In [9]: t_start = time.time()
resultat = gumshap.conditional(train, plot=False,plot_importance=True,
↪percentage=False)
print(f'Run Time : {time.time()-t_start} sec')
```

```
Run Time : 6.594994783401489 sec
```

```
nbsphinx-code-borderwhite
```

```
In [10]: resultat = gumshap.conditional(train, plot=True,plot_importance=False,
↪percentage=False)
```

```
nbsphinx-code-borderwhite
```

The result is returned as a dictionary, the keys are the names of the features and the associated value is the absolute value of the average of the calculated shap.

```
In [11]: t_start = time.time()
resultat = gumshap.conditional(train, plot=False,plot_importance=False,
↪percentage=False)
print(f'Run Time : {time.time()-t_start} sec')
resultat
```

```
Run Time : 6.998995780944824 sec
```

```
Out[11]: {'Z': 0.5464180054433385,
'X2': 0.3271606443752007,
'X1': 0.2533375405370652,
'X0': 0.0617671220000001715,
'X3': 0.10465402104047901,
'R': 0.054456334441524014}
```

## Causal Shap Values

This method is similar to the previous one, except the formula of computation. It computes the causal shap value as described in the paper of Heskes *Causal Shapley Values: Exploiting Causal Knowledge to Explain Individual Predictions of Complex Models*.

```
In [12]: t_start = time.time()
causal = gumshap.causal(train, plot=True, plot_importance=True, percentage=False)
print(f'Run Time : {time.time()-t_start} sec')
```

```
Run Time : 7.8449952602386475 sec
```

nbsphinx-code-borderwhite

As you can see, since  $R$  is not among the ‘causes’ of  $Y$ , its causal importance is null.

## Marginal Shap Values

Similarly, one can also compute marginal Shap Value.

```
In [13]: t_start = time.time()
marginal = gumshap.marginal(train, sample_size=10, plot=True, plot_importance=True,
    ↪percentage=False)
print(f'Run Time : {time.time()-t_start} sec')
print(marginal)
```

```
Run Time : 42.12499499320984 sec
```

```
{'Z': 0.7937690075288257, 'X2': 0.36899838317867956, 'X1': 0.3507606787561853, 'X0': 0.0,
    ↪'X3': 0.0, 'R': 0.0}
```

nbsphinx-code-borderwhite

As you can see, since  $R$ ,  $X0$  and  $X3$  are no in the Markov Blanket of  $Y$ , their marginal importances are null.

## Visualizing shapvalues directly on a BN

This method returns a coloured graph that makes it easier to understand which variable is important and where it is located in the graph.

```
In [14]: import pyAgrum.lib.notebook as gnb

g = gumshap.showShapValues(causal)
gnb.showGraph(g)
```

nbsphinx-code-borderwhite



## Visualizing information

```
In [15]: expl.showInformation(bn)

<IPython.core.display.HTML object>
```

```
In [ ]:
```

## 1.27.4 Kullback-Leibler for Bayesian networks

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrum.org"> (https://agrum.org)</a>	<a href="https://agrum.gitlab.io/extra/agrum_at_binder.html"> (https://agrum.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

```
In [1]: import os

%matplotlib inline

from pylab import *
import matplotlib.pyplot as plt
```

### Initialisation

- importing pyAgrum
- importing pyAgrum.lib tools
- loading a BN

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
```

### Create a first BN : bn

```
In [3]: bn=gum.loadBN("res/asia.bif")
# randomly re-generate parameters for every Conditional Probability Table
bn.generateCPTs()
bn
```

```
Out[3]: (pyAgrum.BayesNet<double>@0000001FE27031DE0) BN{nodes: 8, arcs: 8, domainSize: 256,
↳ dim: 36}
```

### Create a second BN : bn2

```
In [4]: bn2=gum.loadBN("res/asia.bif")
bn2.generateCPTs()
bn2
```

```
Out[4]: (pyAgrum.BayesNet<double>@0000001FE27035300) BN{nodes: 8, arcs: 8, domainSize: 256,
↳ dim: 36}
```

### bn vs bn2 : different parameters

```
In [5]: gnb.flow.row(bn.cpt(3),bn2.cpt(3),
               captions=["a CPT in bn","same CPT in bn2 (with different parameters)"])
<IPython.core.display.HTML object>
```

### Exact and (Gibbs) approximated KL-divergence

In order to compute KL-divergence, we just need to be sure that the 2 distributions are defined on the same domain (same variables, etc.)

Exact KL

```
In [6]: g1=gum.ExactBNdistance(bn,bn2)
        print(g1.compute())

{'klPQ': 5.0030971388003485, 'errorPQ': 0, 'klQP': 3.527314407069579, 'errorQP': 0,
  ↳ 'hellinger': 1.037499761443752, 'bhattacharya': 0.7726296131290553, 'jensen-shannon
  ↳ ': 0.6385791723188207}
```

If the models are not on the same domain :

```
In [7]: bn_different_domain=gum.loadBN("res/alarm.dsl")

# g=gum.BruteForceKL(bn,bn_different_domain) # a KL-divergence between asia and alarm_
↳ ... :(
#
# would cause
#-----
#OperationNotAllowed                                Traceback (most recent call last)
#
#OperationNotAllowed: this operation is not allowed : KL : the 2 BNs are not_
↳ compatible (not the same vars : visit_to_Asia?)
```

Gibbs-approximated KL

```
In [8]: g=gum.GibbsBNdistance(bn,bn2)
        g.setVerbosity(True)
        g.setMaxTime(120)
        g.setBurnIn(5000)
        g.setEpsilon(1e-7)
        g.setPeriodSize(500)
```

```
In [9]: print(g.compute())
        print("Computed in {0} s".format(g.currentTime()))

{'klPQ': 4.999350547660979, 'errorPQ': 0, 'klQP': 3.309147414499417, 'errorQP': 0,
  ↳ 'hellinger': 1.0241904644545237, 'bhattacharya': 0.7786179301380848, 'jensen-shannon
  ↳ ': 0.6228385613041574}
Computed in 1.9164245 s
```

```
In [10]: print("--")

        print(g.messageApproximationScheme())
        print("--")
```

(continues on next page)

(continued from previous page)

```
print("Temps de calcul : {}".format(g.currentTime()))
print("Nombre d'itérations : {}".format(g.nbrIterations()))

--
stopped with epsilon=1e-07
--
Temps de calcul : 1.9164245
Nombre d'itérations : 176500
```

```
In [11]: p=plot(g.history(), 'g')
          nbsphinx-code-borderwhite
```

## Animation of Gibbs KL

Since it may be difficult to know what happens during approximation algorithm, pyAgrum allows to follow the iteration using animated matplotlib figure

```
In [12]: g=gum.GibbsBNdistance(bn,bn2)
g.setMaxTime(60)
g.setBurnIn(500)
g.setEpsilon(1e-7)
g.setPeriodSize(5000)
```

```
In [13]: gnb.animApproximationScheme(g) # logarithmique scale for Y
         g.compute()
```

```
Out[13]: {'klPQ': 5.016855829234866,
          'errorPQ': 0,
          'klQP': 3.3839420489270537,
          'errorQP': 0,
          'hellinger': 1.0291151959143872,
          'bhattacharya': 0.7805282966474169,
          'jensen-shannon': 0.6282516385879086}
```

In [ ]:

### 1.27.5 Comparing BNs



(<http://creativecommons.org/licenses/by-nc/4.0/>)



(<https://agrum.org>)

([https://agrum.gitlab.io/extra/agrum\\_at\\_binder.html](https://agrum.gitlab.io/extra/agrum_at_binder.html))

```
In [1]: def dict2html(di1,di2=None):
        res= "<br/>".join([f"<b>{k:15}</b>:{v}" for k,v in di1.items()])
        if di2 is not None:
            res+="  
<br/>"
```

(continues on next page)

(continued from previous page)

```
res+= "<br/>".join([f"<b>{k:15}</b>: {v}" for k,v in di2.items()])
return res
```

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.lib.bn_vs_bn as gcm
```

## How to compare two BNs

PyAgrum allows you to compare BNs in several ways. This notebook show you some of them: - a graphical diff between the 2 BNs - some scores form recal and precision - distance measures (for more, see notebook 26-klForBNs for more)

### Between two different structures

```
In [3]: bn1=gum.fastBN("A->B->C->D->E<-A->F")
bn2=gum.fastBN("A->B<-C->D->E<-A;F->E")
cmp=gcm.GraphicalBNComparator(bn1,bn2)
kl=gum.ExactBNdistance(bn1,bn2) # bruteForce is possible car the BNs are small
gnb.sideBySide(bn1,bn2,gnb.getBNDiff(bn1,bn2),dict2html(cmp.scores(),cmp.hamming()),
→cmp.equivalentBNs(),dict2html(kl.compute()),
    captions=['bn1','bn2','graphical diff','Scores','equivalent ?'],
→'distances'],valign="bottom")

<IPython.core.display.HTML object>
```

The logic for the arcs of the graphical diff is the following. When comparaing bn1 with bn2 (in that order) : - full black line: the arc is common for both - full red line: the arc is common but inverted in bn2 - dotted black line: the arc is added in bn2 - dotted red line: the arc is removed in bn2

For the scores : - precision and recall are computed considering BN1 as the reference -  $Fscore = \frac{2 \cdot recall \cdot precision}{recall + precision}$  is the weighted average of Precision and Recall. -  $dist2opt = \sqrt{(1 - precision)^2 + (1 - recall)^2}$  represents the euclidian distance to the ideal(precision=1,recall=1)

EquivalentBN return “OK” if equivalent or a reason for non equivalence

Finally, BruteForceKL compute in the same time several distances : I-projection, M-projection, Hellinger and Bhattacharya. For more complex BNs, there exists a GibbsKL to approximate those distances. Of course, the computation are much slower.

### Same structure, different parameters

```
In [4]: bn1=gum.fastBN("A->B->C->D->E<-A->F")
bn2=gum.fastBN("A->B->C->D->E<-A->F")
cmp=gcm.GraphicalBNComparator(bn1,bn2)
kl=gum.ExactBNdistance(bn1,bn2) # bruteForce is possible car the BNs are small
gnb.sideBySide(bn1,bn2,gnb.getBNDiff(bn1,bn2),dict2html(cmp.scores(),cmp.hamming()),
→cmp.equivalentBNs(),dict2html(kl.compute()),
    captions=['bn1','bn2','graphical diff','Scores','equivalent ?'],
→'distances'],valign="bottom")

<IPython.core.display.HTML object>
```

## identical BNs

```
In [5]: bn1=gum.fastBN("A->B->C->D->E<-A->F")
bn2=bn1
cmp=gcm.GraphicalBNComparator(bn1,bn2)
kl=gum.ExactBNdistance(bn1,bn2) # bruteForce is possible car the BNs are small
gnb.sideBySide(bn1,bn2,gnb.getBNDiff(bn1,bn2),dict2html(cmp.scores(),cmp.hamming()),
↳cmp.equivalentBNs(),dict2html(kl.compute()),
    captions=['bn1','bn2','graphical diff','Scores','equivalent ?'],
↳'distances'],valign="bottom")



<IPython.core.display.HTML object>
```

In the notebook `Learning_DirichletPriorAndWeightedDatabase`, you can find an interesting discussion on how can change those scores and distance.

```
In [ ]:
```

```
In [ ]:
```

## 1.27.6 Coloring and exporting graphical models as image (pdf, png)

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org">           (https://agrums.org)         </a>	<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html">           (https://agrums.gitlab.io/extra/agrum_at_binder.html)         </a>
--	---	---

```
In [1]: from pylab import *
import matplotlib.pyplot as plt
```

```
In [2]: import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

bn=gum.fastBN("a->b->c->d;b->e->d->f;g->c")
gnb.flow.row(bn,gnb.getInference(bn))

<IPython.core.display.HTML object>
```

## customizing colours and width for model and inference

```
In [3]: def nodevalue(n):
    return 0.5 if n in "aeiou" else 0.7
def arcvalue(a):
    return (10-a[0])*a[1]
def arcvalue2(a):
    return (a[0]+a[1]+5)/22
gnb.showBN(bn,
    nodeColor={n:nodevalue(n) for n in bn.names()},
```

(continues on next page)

(continued from previous page)

```
arcWidth={a:arcvalue(a) for a in bn.arcs()},
arcColor={a:arcvalue2(a) for a in bn.arcs()]}
```

nbsphinx-code-borderwhite

```
In [4]: gnb.showInference(bn,
    targets={"a", "g", "f", "b"},
    evs={'e':0},
    nodeColor={n:nodevalue(n) for n in bn.names()},
    arcWidth={a:arcvalue(a) for a in bn.arcs()})
```

nbsphinx-code-borderwhite

```
In [5]: gnb.flow.row(gnb.getBN(bn,
    nodeColor={n:nodevalue(n) for n in bn.names()},
    arcWidth={a:arcvalue(a) for a in bn.arcs()}),
    gnb.getInference(bn,
    nodeColor={n:nodevalue(n) for n in bn.names()},
    arcWidth={a:arcvalue(a) for a in bn.arcs()}))

)
```

&lt;IPython.core.display.HTML object&gt;

```
In [6]: import matplotlib.pyplot as plt
mycmap=plt.get_cmap('Reds')
formyarcs=plt.get_cmap('winter')
gnb.flow.row(gnb.getBN(bn,
    nodeColor={n:nodevalue(n) for n in bn.names()},
    arcColor={a:arcvalue2(a) for a in bn.arcs()},
    cmap=mycmap,
    cmapArc=formyarcs),
    gnb.getInference(bn,
    nodeColor={n:nodevalue(n) for n in bn.names()},
    arcColor={a:arcvalue2(a) for a in bn.arcs()},
    arcWidth={a:arcvalue(a) for a in bn.arcs()},
    cmap=mycmap,
    cmapArc=formyarcs))

)
```

&lt;IPython.core.display.HTML object&gt;

## Exporting model and inference as image

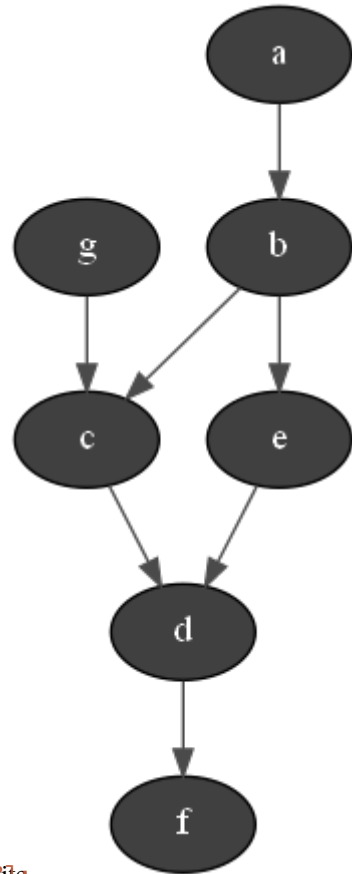
Exporting as image (pdf, png, etc.) has been gathered in 2 functions : `pyAgrum.lib.image.export()` and `pyAgrum.lib.image.exportInference()`. The argument are the same as for `pyAgrum.notebook.show{Model}` and `pyAgrum.notebook.show{Inference}`.

```
In [7]: import pyAgrum.lib.image as gumimage
from IPython.display import Image # to display the exported images
```

```
In [8]: gumimage.export(bn, "out/test_export.png")
```

```
Image(filename='out/test_export.png')
```





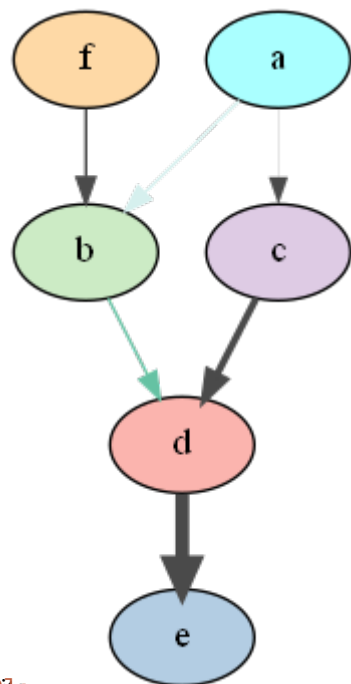
nbsphinx-code-border [1]

```

In [9]: bn = gum.fastBN("a->b->d;a->c->d[3]->e;f->b")
gumimage.export(bn,"out/test_export.png",
                nodeColor={'a': 1,
                           'b': 0.3,
                           'c': 0.4,
                           'd': 0.1,
                           'e': 0.2,
                           'f': 0.5},
                arcColor={(0, 1): 0.2,
                          (1, 2): 0.5},
                arcWidth={(0, 3): 0.4,
                          (3, 2): 0.5,
                          (2,4) :0.6})

Image(filename='out/test_export.png')

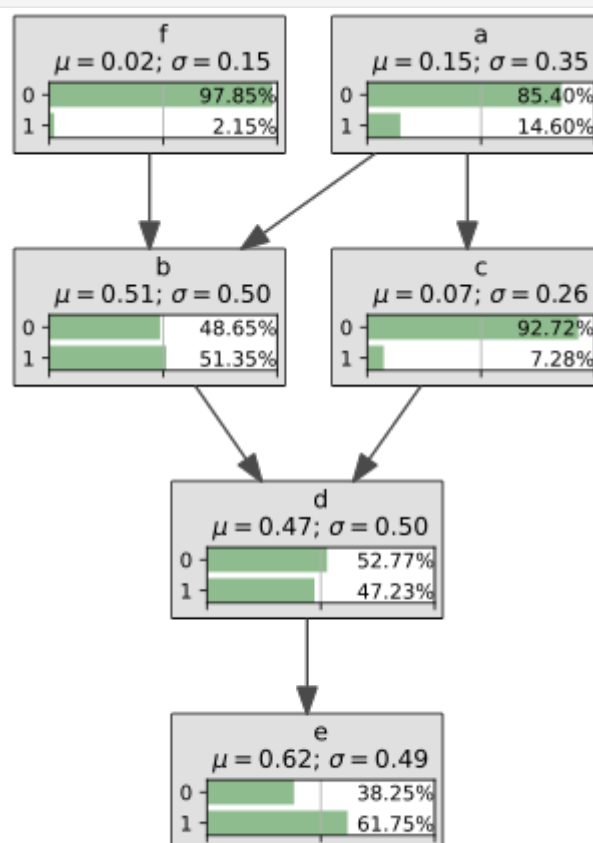
```



nbsphinx-code-border style

```
In [10]: gumimage.exportInference(bn, "out/test_export.png")
```

```
Image(filename='out/test_export.png')
```



Inference in 1.00ms

nbsphinx-code-border style

```
In [11]: gumimage.export(bn, "out/test_export.pdf")
```

[Link to out/test\\_export.pdf](#)

### exporting inference with evidence

```
In [12]: bn=gum.loadBN("res/alarm.dsl")
gumimage.exportInference(bn, "out/test_export.pdf",
                        evs={"CO":1, "VENTLUNG":1},
                        targets={"VENTALV",
                                "CATECHOL",
                                "HR",
                                "MINVOLSET",
                                "ANAPHYLAXIS",
                                "STROKEVOLUME",
                                "ERRLOWOUTPUT",
                                "HBR",
                                "PULMEMBOLUS",
                                "HISTORY",
                                "BP",
                                "PRESS",
                                "CO"},
                        size="15!")
```

[Link to out/test\\_export.pdf](#)

### Other models

Other models can also use these functions.

```
In [13]: infdiag=gum.loadID("res/OilWildcatter.bifxml")
gumimage.export(infdiag, "out/test_export.pdf")
```

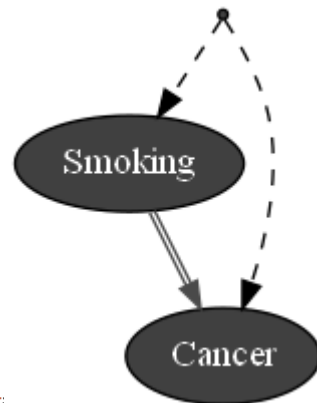
[Link to out/test\\_export.pdf](#)

```
In [14]: gumimage.exportInference(infdiag, "out/test_export.pdf")
```

[Link to out/test\\_export.pdf](#)

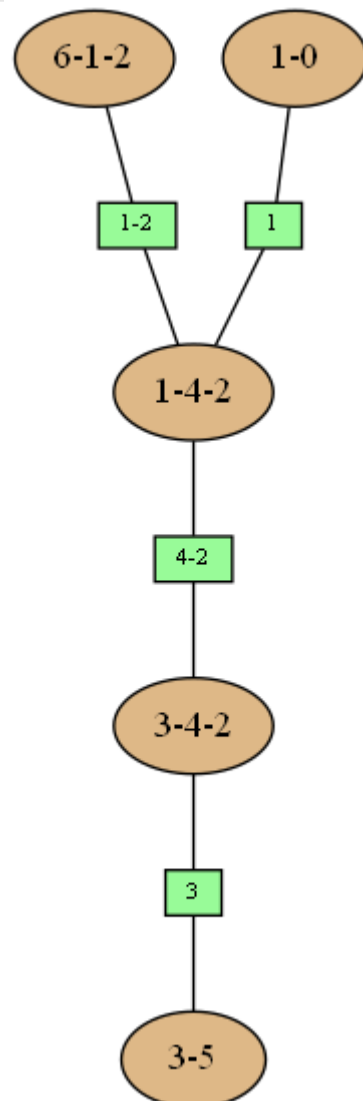
### Exporting any object with toDot() method

```
In [15]: import pyAgrum.causal as csl
obs1 = gum.fastBN("Smoking->Cancer")
modele3 = csl.CausalModel(obs1, [("Genotype", ["Smoking", "Cancer"])], True)
gumimage.export(modele3, "out/test_export.png") # a causal model has a toDot method.
Image(filename='out/test_export.png')
```



nbsphinx-code-block-style

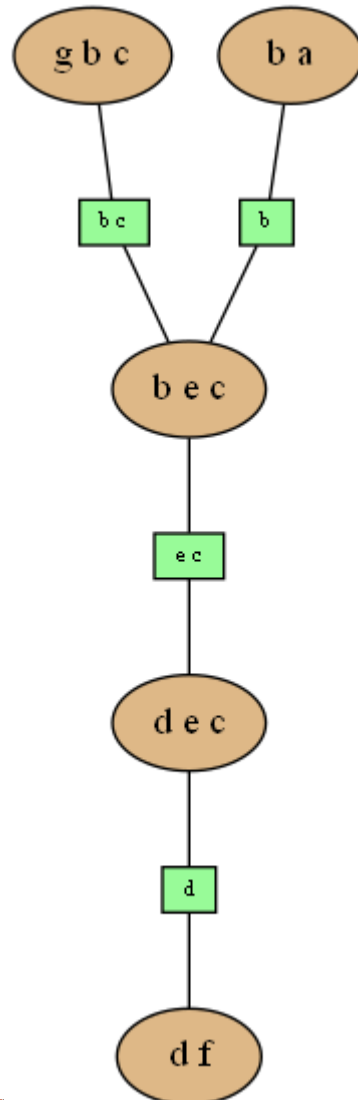
```
In [16]: bn=gum.fastBN("a->b->c->d;b->e->d->f;g->c")
ie=gum.LazyPropagation(bn)
jt=ie.junctionTree()
gumimage.export(jt,"out/test_export.png") # a JunctionTree has a method jt.toDot()
Image(filename='out/test_export.png')
```



nbsphinx-code-block-style

... or even a string in dot syntax

```
In [17]: gumimage.export(jt.toDotWithNames(bn), "out/test_export.png") # jt.toDotWithNames(bn)
↳ creates a dot-string for a junction tree with names of variables
Image(filename='out/test_export.png')
```



nbsphinx-code-border style="border: 1px solid #ccc; padding: 10px; margin: 10px 0;">

## Exporting to pyplot

```
In [18]: import matplotlib.pyplot as plt

bn=gum.fastBN("A->B->C<-D")

plt.imshow(gumimage.export(bn))
plt.show()

plt.imshow(gumimage.exportInference(bn,size="15!"))
plt.show()

plt.figure(figsize = (10,10))
plt.imshow(gumimage.exportInference(bn,size="15!"))
```



(continues on next page)

(continued from previous page)

```
plt.show()
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
nbsphinx-code-borderwhite
```

In [ ]:

### 1.27.7 gum.config :the configuration object for pyAgrum

 <a href="http://creativecommons.org/licenses/by-nc/4.0/">(http://creativecommons.org/licenses/by-nc/4.0/)</a>	 <a href="https://agrums.org"> (https://agrums.org)</a>	<a href="https://agrums.gitlab.io/extra/agrum_at_binder.html"> (https://agrums.gitlab.io/extra/agrum_at_binder.html)</a>
--	---	--

```
In [1]: import pyAgrum as gum
print("="*35)
print(gum.config)
print("="*35)
```

```
=====
[core]
  default_maxnumberofthreads = 24
[notebook]
  potential_visible_digits = 4
  potential_with_colors = True
  potential_color_0 = #FF7F64
  potential_color_1 = #7FFF64
  potential_with_fraction = False
  potential_fraction_limit = 50
  potential_fraction_round_error = 1e-6
  potential_fraction_with_latex = True
  histogram_horizontal_visible_digits = 2
  histogram_vertical_visible_digits = 2
  histogram_horizontal_threshold = 8
  histogram_line_threshold = 40
  histogram_color = darkseagreen
  histogram_use_percent = True
  histogram_mode = compact
  potential_parent_values = merge
  figure_facecolor = #E0E0E0
  graph_format = svg
  show_inference_time = True
  default_graph_size = 5
  default_graph_inference_size = 8
  default_arc_color = #4A4A4A
  default_node_bgcolor = #404040
  default_node_fgcolor = white
  evidence_bgcolor = sandybrown
  evidence_fgcolor = black
  default_node_cmap = Pastel1
```

(continues on next page)

(continued from previous page)

```

default_arc_cmap = BuGn
default_edge_cmap = BuGn
default_markovnetwork_view = factorgraph
junctiontree_graph_size = 10
junctiontree_with_names = True
junctiontree_separator_bgcolor = palegreen
junctiontree_separator_fgcolor = black
junctiontree_separator_fontsize = 8
junctiontree_clique_bgcolor = burlywood
junctiontree_clique_fgcolor = black
junctiontree_clique_fontsize = 10
junctiontree_map_cliquescale = 0.3
junctiontree_map_sepscale = 0.1
junctiontree_map_edgelen = 1
graphdiff_missing_style = dashed
graphdiff_missing_color = red
graphdiff_overflow_style = dashed
graphdiff_overflow_color = purple
graphdiff_reversed_style = solid
graphdiff_reversed_color = purple
graphdiff_correct_style = solid
graphdiff_correct_color = grey
[factorgraph]
default_node_bgcolor = coral
default_node_fgcolor = black
default_factor_bgcolor = burlywood
edge_length = 0.7
edge_length_inference = 0.9
[dynamicBN]
default_graph_size = 6
[influenceDiagram]
default_graph_size = 6
default_chance_bgcolor = #808080
default_chance_fgcolor = white
default_utility_bgcolor = #50508A
default_utility_fgcolor = white
default_decision_bgcolor = #9A5050
default_decision_fgcolor = white
chance_shape = ellipse
utility_shape = hexagon
decision_shape = box
decision_arc_style = tapered, bold, dotted
utility_arc_style = dashed
default_id_size = 6
default_id_inference_size = 6
utility_visible_digits = 2
utility_show_stdev = True
utility_show_loss = False
[credalnet]
default_node_bgcolor = #404040
default_node_fgcolor = white
histo_max_color = #BBFFAA
[causal]
show_latent_names = False
latex_do_prefix = \hookrightarrow\mkern-6.5mu
latex_do_suffix =

```

(continues on next page)

(continued from previous page)

```
default_graph_size = 2.5
default_node_bgcolor = #404040
default_node_fgcolor = white
default_latent_bgcolor = #A08080
default_latent_fgcolor = black
[ROC]
draw_color = #008800
fill_color = #AAEEAA
=====
```

### `gum.config` as singleton

As far as it can be done with Python, `gum.config` is a singleton

```
In [2]: cfg2=gum.PyAgrumConfiguration()
print(id(cfg2))
print(id(gum.config))
```

```
1552049215552
1552049215552
```

### `pyagrums.ini`

The configuration is a mix between the defaults (which exactly define the possible section and property) and a file `pyagrums.ini` (if present in the current folder) that contains only the changed properties.

section and key are case insensitive.

### `config` constantly keeps track of the differences between defaults and actual values :

```
In [3]: gum.config['dynamicBN','default_graph_size']=10

print("Configuration only for the current session")
print('-'*40)
gum.config.diff()
```

```
Configuration only for the current session
-----
[dynamicBN]
default_graph_size = 10
```

```
In [4]: # if there is a local modification, __repr__ shows only the diff other wise it show
↳ all properties
gum.config
```

```
Out[4]: [dynamicBN]
default_graph_size = 10
```

```
In [5]: # __str__ shows all the properties
print(gum.config)
```

```
[core]
default_maxnumberofthreads = 24
[notebook]
potential_visible_digits = 4
```

(continues on next page)



(continued from previous page)

```

potential_with_colors = True
potential_color_0 = #FF7F64
potential_color_1 = #7FFF64
potential_with_fraction = False
potential_fraction_limit = 50
potential_fraction_round_error = 1e-6
potential_fraction_with_latex = True
histogram_horizontal_visible_digits = 2
histogram_vertical_visible_digits = 2
histogram_horizontal_threshold = 8
histogram_line_threshold = 40
histogram_color = darkseagreen
histogram_use_percent = True
histogram_mode = compact
potential_parent_values = merge
figure_facecolor = #E0E0E0
graph_format = svg
show_inference_time = True
default_graph_size = 5
default_graph_inference_size = 8
default_arc_color = #4A4A4A
default_node_bgcolor = #404040
default_node_fgcolor = white
evidence_bgcolor = sandybrown
evidence_fgcolor = black
default_node_cmap = Pastel1
default_arc_cmap = BuGn
default_edge_cmap = BuGn
default_markovnetwork_view = factorgraph
junctiontree_graph_size = 10
junctiontree_with_names = True
junctiontree_separator_bgcolor = palegreen
junctiontree_separator_fgcolor = black
junctiontree_separator_fontsize = 8
junctiontree_clique_bgcolor = burlywood
junctiontree_clique_fgcolor = black
junctiontree_clique_fontsize = 10
junctiontree_map_cliquescale = 0.3
junctiontree_map_sepscale = 0.1
junctiontree_map_edgelen = 1
graphdiff_missing_style = dashed
graphdiff_missing_color = red
graphdiff_overflow_style = dashed
graphdiff_overflow_color = purple
graphdiff_reversed_style = solid
graphdiff_reversed_color = purple
graphdiff_correct_style = solid
graphdiff_correct_color = grey
[factorgraph]
default_node_bgcolor = coral
default_node_fgcolor = black
default_factor_bgcolor = burlywood
edge_length = 0.7
edge_length_inference = 0.9
[dynamicBN]
default_graph_size = 10

```

(continues on next page)

(continued from previous page)

```
[influenceDiagram]
    default_graph_size = 6
    default_chance_bgcolor = #808080
    default_chance_fgcolor = white
    default_utility_bgcolor = #50508A
    default_utility_fgcolor = white
    default_decision_bgcolor = #9A5050
    default_decision_fgcolor = white
    chance_shape = ellipse
    utility_shape = hexagon
    decision_shape = box
    decision_arc_style = tapered, bold, dotted
    utility_arc_style = dashed
    default_id_size = 6
    default_id_inference_size = 6
    utility_visible_digits = 2
    utility_show_stdev = True
    utility_show_loss = False
[credalnet]
    default_node_bgcolor = #404040
    default_node_fgcolor = white
    histo_max_color = #BBFFAA
[causal]
    show_latent_names = False
    latex_do_prefix = \hookrightarrow\mkern-6.5mu
    latex_do_suffix =
    default_graph_size = 2.5
    default_node_bgcolor = #404040
    default_node_fgcolor = white
    default_latent_bgcolor = #A08080
    default_latent_fgcolor = black
[ROC]
    draw_color = #008800
    fill_color = #AAEEAA
```

## Accessors

### Getter

```
In [6]: print(gum.config["notebook", "evidence_bgcolor"])
print(gum.config.get("notebook", "evidence_bgcolor"))

sandybrown
sandybrown
```

## Setter

```
In [7]: gum.config["notebook","evidence_bgcolor"]="papayawhip"
gum.config["causal","default_graph_size"]=1.9
```

```
In [8]: gum.config # once again, only the diff with defaults
```

```
Out[8]: [notebook]
        evidence_bgcolor = papayawhip
[dynamicBN]
        default_graph_size = 10
[causal]
        default_graph_size = 1.9
```

## constant structure, mutable content

The structure section.key is fixed by default and readonly : one can only change the value of an existing property

```
In [9]: try:
        gum.config["AAA","000"]=1
except SyntaxError as e:
    print("Syntax error : {}".format(e.msg))

Syntax error : You can not add section 'AAA' in pyAgrum configuration
```

```
In [10]: try:
        gum.config["causal","000"]=1
except SyntaxError as e:
    print("Syntax error : {}".format(e.msg))

Syntax error : You can not add option 'causal,000' in pyAgrum configuration
```

## properties as string

All the properties are stored as string !

```
In [11]: gum.config["notebook","default_graph_size"]

Out[11]: '5'
```

```
In [12]: gum.config["notebook","default_graph_size"]=10
gum.config["notebook","default_graph_size"]

Out[12]: '10'
```

## Reset, reload, save

The configuration can be saved in the current folder (gum.config.save()).

The configuration can be restored from the default (reset) or the current saved state (reload).

```
In [13]: gum.config.reset() # back to defaults
gum.config # no diff => shows all the properties
```

```
Out[13]: # no customized property
[core]
    default_maxnumberofthreads = 24
[notebook]
    potential_visible_digits = 4
    potential_with_colors = True
    potential_color_0 = #FF7F64
    potential_color_1 = #7FFF64
    potential_with_fraction = False
    potential_fraction_limit = 50
    potential_fraction_round_error = 1e-6
    potential_fraction_with_latex = True
    histogram_horizontal_visible_digits = 2
    histogram_vertical_visible_digits = 2
    histogram_horizontal_threshold = 8
    histogram_line_threshold = 40
    histogram_color = darkseagreen
    histogram_use_percent = True
    histogram_mode = compact
    potential_parent_values = merge
    figure_facecolor = #E0E0E0
    graph_format = svg
    show_inference_time = True
    default_graph_size = 5
    default_graph_inference_size = 8
    default_arc_color = #4A4A4A
    default_node_bgcolor = #404040
    default_node_fgcolor = white
    evidence_bgcolor = sandybrown
    evidence_fgcolor = black
    default_node_cmap = Pastel1
    default_arc_cmap = BuGn
    default_edge_cmap = BuGn
    default_markovnetwork_view = factorgraph
    junctiontree_graph_size = 10
    junctiontree_with_names = True
    junctiontree_separator_bgcolor = palegreen
    junctiontree_separator_fgcolor = black
    junctiontree_separator_fontsize = 8
    junctiontree_clique_bgcolor = burlywood
    junctiontree_clique_fgcolor = black
    junctiontree_clique_fontsize = 10
    junctiontree_map_cliquescale = 0.3
    junctiontree_map_sepscale = 0.1
    junctiontree_map_edgelen = 1
    graphdiff_missing_style = dashed
    graphdiff_missing_color = red
    graphdiff_overflow_style = dashed
    graphdiff_overflow_color = purple
    graphdiff_reversed_style = solid
    graphdiff_reversed_color = purple
    graphdiff_correct_style = solid
    graphdiff_correct_color = grey
[factorgraph]
    default_node_bgcolor = coral
    default_node_fgcolor = black
    default_factor_bgcolor = burlywood
```

(continues on next page)

(continued from previous page)

```

    edge_length = 0.7
    edge_length_inference = 0.9
[dynamicBN]
    default_graph_size = 6
[influenceDiagram]
    default_graph_size = 6
    default_chance_bgcolor = #808080
    default_chance_fgcolor = white
    default_utility_bgcolor = #50508A
    default_utility_fgcolor = white
    default_decision_bgcolor = #9A5050
    default_decision_fgcolor = white
    chance_shape = ellipse
    utility_shape = hexagon
    decision_shape = box
    decision_arc_style = tapered, bold, dotted
    utility_arc_style = dashed
    default_id_size = 6
    default_id_inference_size = 6
    utility_visible_digits = 2
    utility_show_stdev = True
    utility_show_loss = False
[credalnet]
    default_node_bgcolor = #404040
    default_node_fgcolor = white
    histo_max_color = #BBFFAA
[causal]
    show_latent_names = False
    latex_do_prefix = \hookrightarrow\mkern-6.5mu
    latex_do_suffix =
    default_graph_size = 2.5
    default_node_bgcolor = #404040
    default_node_fgcolor = white
    default_latent_bgcolor = #A08080
    default_latent_fgcolor = black
[ROC]
    draw_color = #008800
    fill_color = #AAEEAA

```

```

In [14]: try:
        gum.config.load() # reload pyagrums.ini
except FileNotFoundError:
    pass # no pyagrums.ini in the folder
gum.config

```

```

Out[14]: # no customized property
[core]
    default_maxnumberofthreads = 24
[notebook]
    potential_visible_digits = 4
    potential_with_colors = True
    potential_color_0 = #FF7F64
    potential_color_1 = #7FFF64
    potential_with_fraction = False
    potential_fraction_limit = 50
    potential_fraction_round_error = 1e-6

```

(continues on next page)

(continued from previous page)

```

potential_fraction_with_latex = True
histogram_horizontal_visible_digits = 2
histogram_vertical_visible_digits = 2
histogram_horizontal_threshold = 8
histogram_line_threshold = 40
histogram_color = darkseagreen
histogram_use_percent = True
histogram_mode = compact
potential_parent_values = merge
figure_facecolor = #E0E0E0
graph_format = svg
show_inference_time = True
default_graph_size = 5
default_graph_inference_size = 8
default_arc_color = #4A4A4A
default_node_bgcolor = #404040
default_node_fgcolor = white
evidence_bgcolor = sandybrown
evidence_fgcolor = black
default_node_cmap = Pastel1
default_arc_cmap = BuGn
default_edge_cmap = BuGn
default_markovnetwork_view = factorgraph
junctiontree_graph_size = 10
junctiontree_with_names = True
junctiontree_separator_bgcolor = palegreen
junctiontree_separator_fgcolor = black
junctiontree_separator_fontsize = 8
junctiontree_clique_bgcolor = burlywood
junctiontree_clique_fgcolor = black
junctiontree_clique_fontsize = 10
junctiontree_map_cliquescale = 0.3
junctiontree_map_sepscale = 0.1
junctiontree_map_edgelen = 1
graphdiff_missing_style = dashed
graphdiff_missing_color = red
graphdiff_overflow_style = dashed
graphdiff_overflow_color = purple
graphdiff_reversed_style = solid
graphdiff_reversed_color = purple
graphdiff_correct_style = solid
graphdiff_correct_color = grey
[factorgraph]
default_node_bgcolor = coral
default_node_fgcolor = black
default_factor_bgcolor = burlywood
edge_length = 0.7
edge_length_inference = 0.9
[dynamicBN]
default_graph_size = 6
[influenceDiagram]
default_graph_size = 6
default_chance_bgcolor = #808080
default_chance_fgcolor = white
default_utility_bgcolor = #50508A
default_utility_fgcolor = white

```

(continues on next page)

(continued from previous page)

```

default_decision_bgcolor = #9A5050
default_decision_fgcolor = white
chance_shape = ellipse
utility_shape = hexagon
decision_shape = box
decision_arc_style = tapered, bold, dotted
utility_arc_style = dashed
default_id_size = 6
default_id_inference_size = 6
utility_visible_digits = 2
utility_show_stdev = True
utility_show_loss = False
[credalnet]
default_node_bgcolor = #404040
default_node_fgcolor = white
histo_max_color = #BBFFAA
[causal]
show_latent_names = False
latex_do_prefix = \hookrightarrow\mkern-6.5mu
latex_do_suffix =
default_graph_size = 2.5
default_node_bgcolor = #404040
default_node_fgcolor = white
default_latent_bgcolor = #A08080
default_latent_fgcolor = black
[ROC]
draw_color = #008800
fill_color = #AAEEAA

```

## Using configuration

In [15]: `import pyAgrum.lib.notebook as gnb`

```
bn=gum.fastBN("D->C<-A->B[4];A->E")
bn.cpt("B")
```

Out[15]: (pyAgrum.Potential<double>@000001695C982BD0)

	B				
A	0	1	2	3	
0	0.0908	0.6597	0.2455	0.0040	
1	0.0527	0.2998	0.4116	0.2360	

In [16]: `gum.config["notebook", "potential_visible_digits"]=1`  
`bn.cpt("B")`

Out[16]: (pyAgrum.Potential<double>@000001695C982BD0)

	B				
A	0	1	2	3	
0	0.0908	0.6597	0.2455	0.0040	
1	0.0527	0.2998	0.4116	0.2360	

In [17]: `gum.config['notebook', 'potential_color_0']="#AA00AA"`  
`gum.config['notebook', 'potential_color_1']="#00FFAA"`

(continues on next page)

(continued from previous page)

bn.cpt("B")

Out [17]: (pyAgrum.Potential&lt;double&gt;@0000001695C982BD0)

	B				
A	0	1	2	3	
0	0.0908	0.6597	0.2455	0.0040	
1	0.0527	0.2998	0.4116	0.2360	

```
In [18]: gum.config["notebook", "potential_visible_digits"]=4
gnb.flow.add(bn.cpt("B"), "Ugly float")

gum.config["notebook", "potential_visible_digits"]=1
gnb.flow.add(bn.cpt("B"), "Ugly 1digit-float")

gum.config['notebook', 'potential_with_fraction']=True
gum.config['notebook', 'potential_fraction_with_latex']=False
gum.config['notebook', 'potential_fraction_limit']=2000
gnb.flow.add(bn.cpt("B"), "Simple fraction")

gum.config['notebook', 'potential_fraction_with_latex']=True
gnb.flow.add(bn.cpt("B"), "Sophisticated fraction with LaTeX")
gnb.flow.display()
```

&lt;IPython.core.display.HTML object&gt;

In [19]: gnb.sideBySide(bn, gnb.getInference(bn, evs={"A": 1}, targets={"B"}))

&lt;IPython.core.display.HTML object&gt;

```
In [20]: gum.config["notebook", "evidence_bgcolor"]="green"
gum.config["notebook", "default_node_bgcolor"]="yellow"
gum.config["notebook", "default_node_fgcolor"]="red"
gnb.sideBySide(bn, gnb.getInference(bn, evs={"A": 1}, targets={"B"}))
```

&lt;IPython.core.display.HTML object&gt;

```
In [21]: gum.config["notebook", "default_graph_size"]=1
gnb.sideBySide(bn, gnb.getInference(bn, evs={"A": 1}, targets={"B"}))
```

&lt;IPython.core.display.HTML object&gt;

```
In [22]: gum.config["notebook", "default_graph_inference_size"]="1"
gnb.sideBySide(bn, gnb.getInference(bn, evs={"A": 1}, targets={"B"}))
```

&lt;IPython.core.display.HTML object&gt;

## Finding a specific property

```
In [23]: #find anything containing arc
gum.config.grep("arc")
```

```
[notebook]
  default_arc_color = #4A4A4A
  default_arc_cmap = BuGn
[influenceDiagram]
```

(continues on next page)



(continued from previous page)

```
decision_arc_style = tapered, bold, dotted
utility_arc_style = dashed
```

In [24]: *#find anything containing default*  
 gum.config.grep("default")

```
[core]
  default_maxnumberofthreads = 24
[notebook]
  default_graph_size = 1
  default_graph_inference_size = 1
  default_arc_color = #4A4A4A
  default_node_bgcolor = yellow
  default_node_fgcolor = red
  default_node_cmap = Pastel1
  default_arc_cmap = BuGn
  default_edge_cmap = BuGn
  default_markovnetwork_view = factorgraph
[factorgraph]
  default_node_bgcolor = coral
  default_node_fgcolor = black
  default_factor_bgcolor = burlywood
[dynamicBN]
  default_graph_size = 6
[influenceDiagram]
  default_graph_size = 6
  default_chance_bgcolor = #808080
  default_chance_fgcolor = white
  default_utility_bgcolor = #50508A
  default_utility_fgcolor = white
  default_decision_bgcolor = #9A5050
  default_decision_fgcolor = white
  default_id_size = 6
  default_id_inference_size = 6
[credalnet]
  default_node_bgcolor = #404040
  default_node_fgcolor = white
[causal]
  default_graph_size = 2.5
  default_node_bgcolor = #404040
  default_node_fgcolor = white
  default_latent_bgcolor = #A08080
  default_latent_fgcolor = black
```

In [25]: *# if a section contains the search, all its properties are shown*  
 gum.config.grep("caus")

```
[causal]
  show_latent_names = False
  latex_do_prefix = \hookrightarrow\mkern-6.5mu
  latex_do_suffix =
  default_graph_size = 2.5
  default_node_bgcolor = #404040
  default_node_fgcolor = white
  default_latent_bgcolor = #A08080
  default_latent_fgcolor = black
```

### Saving current configuration in pyagrum.ini

```
In [26]: gum.config.reset() # back to defaults
gum.config['notebook','default_arc_color'] = "#AAAAAA"
gum.config['notebook','evidence_bgcolor'] = "green"
gum.config.save() # store curent changes
```

```
In [27]: gum.config.reset() # back to defaults
gum.config.save() # store defaults back
```

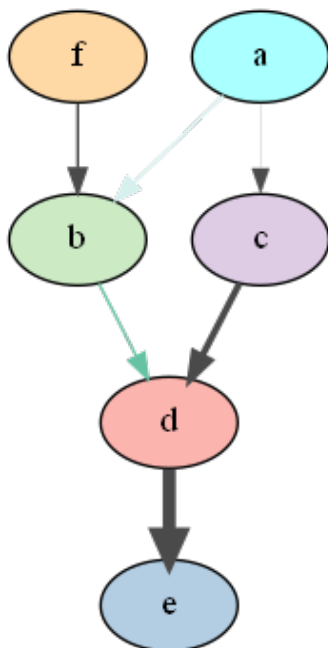
### From PyAgrumConfiguration to ConfigParser

```
In [28]: from configparser import ConfigParser
c=ConfigParser()

gum.config['notebook','default_arc_color'] = "#AAAAAA"
gum.config['notebook','evidence_bgcolor'] = "green"
c.read_string(gum.config.__repr__())
print(c.sections())

['notebook']
```

```
In [ ]:
```



## INDICES AND TABLES

- genindex
- modindex
- search



## PYTHON MODULE INDEX

### p

`pyAgrum.causal.notebook`, [253](#)



## A

- `about()` (in module `pyAgrum`), 278
- `abs()` (`pyAgrum.Potential` method), 54
- `add()` (`pyAgrum.BayesNet` method), 64
- `add()` (`pyAgrum.InfluenceDiagram` method), 190
- `add()` (`pyAgrum.Instantiation` method), 47
- `add()` (`pyAgrum.MarkovNet` method), 217
- `add()` (`pyAgrum.Potential` method), 54
- `add_hook()` (`pyAgrum.PyAgrumConfiguration` method), 292
- `addAllTargets()` (`pyAgrum.GibbsSampling` method), 127
- `addAllTargets()` (`pyAgrum.ImportanceSampling` method), 148
- `addAllTargets()` (`pyAgrum.LazyPropagation` method), 99
- `addAllTargets()` (`pyAgrum.LoopyBeliefPropagation` method), 120
- `addAllTargets()` (`pyAgrum.LoopyGibbsSampling` method), 154
- `addAllTargets()` (`pyAgrum.LoopyImportanceSampling` method), 175
- `addAllTargets()` (`pyAgrum.LoopyMonteCarloSampling` method), 162
- `addAllTargets()` (`pyAgrum.LoopyWeightedSampling` method), 168
- `addAllTargets()` (`pyAgrum.MonteCarloSampling` method), 134
- `addAllTargets()` (`pyAgrum.ShaferShenoyInference` method), 106
- `addAllTargets()` (`pyAgrum.ShaferShenoyMNIInference` method), 223
- `addAllTargets()` (`pyAgrum.VariableElimination` method), 113
- `addAllTargets()` (`pyAgrum.WeightedSampling` method), 141
- `addAMPLITUDE()` (`pyAgrum.BayesNet` method), 64
- `addAND()` (`pyAgrum.BayesNet` method), 64
- `addArc()` (`pyAgrum.BayesNet` method), 64
- `addArc()` (`pyAgrum.CredalNet` method), 204
- `addArc()` (`pyAgrum.DAG` method), 8
- `addArc()` (`pyAgrum.DiGraph` method), 5
- `addArc()` (`pyAgrum.InfluenceDiagram` method), 192
- `addArc()` (`pyAgrum.MixedGraph` method), 20
- `addArcs()` (`pyAgrum.BayesNet` method), 65
- `addArcs()` (`pyAgrum.BayesNetFragment` method), 90
- `addArcs()` (`pyAgrum.InfluenceDiagram` method), 192
- `addCausalArc()` (`pyAgrum.causal.CausalModel` method), 236
- `addChanceNode()` (`pyAgrum.InfluenceDiagram` method), 192
- `addCOUNT()` (`pyAgrum.BayesNet` method), 65
- `addDecisionNode()` (`pyAgrum.InfluenceDiagram` method), 192
- `addEdge()` (`pyAgrum.CliqueGraph` method), 15
- `addEdge()` (`pyAgrum.MixedGraph` method), 20
- `addEdge()` (`pyAgrum.UndiGraph` method), 12
- `addEvidence()` (`pyAgrum.GibbsSampling` method), 127
- `addEvidence()` (`pyAgrum.ImportanceSampling` method), 148
- `addEvidence()` (`pyAgrum.LazyPropagation` method), 99
- `addEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 120
- `addEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 154
- `addEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 175
- `addEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 162
- `addEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 168
- `addEvidence()` (`pyAgrum.MonteCarloSampling` method), 134
- `addEvidence()` (`pyAgrum.ShaferShenoyInference` method), 106
- `addEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 201
- `addEvidence()` (`pyAgrum.ShaferShenoyMNIInference` method), 224
- `addEvidence()` (`pyAgrum.VariableElimination` method), 113
- `addEvidence()` (`pyAgrum.WeightedSampling` method), 141
- `addEXISTS()` (`pyAgrum.BayesNet` method), 65

`addFactor()` (*pyAgrum.MarkovNet* method), 217  
`addFORALL()` (*pyAgrum.BayesNet* method), 65  
`addForbiddenArc()` (*pyAgrum.BNLearner* method), 182  
`addJointTarget()` (*pyAgrum.LazyPropagation* method), 99  
`addJointTarget()` (*pyAgrum.ShaferShenoyInference* method), 107  
`addJointTarget()` (*pyAgrum.ShaferShenoyMNIInference* method), 224  
`addJointTarget()` (*pyAgrum.VariableElimination* method), 114  
`addLabel()` (*pyAgrum.LabelizedVariable* method), 28  
`addLatentVariable()` (*pyAgrum.causal.CausalModel* method), 236  
`addLogit()` (*pyAgrum.BayesNet* method), 65  
`addMandatoryArc()` (*pyAgrum.BNLearner* method), 182  
`addMAX()` (*pyAgrum.BayesNet* method), 66  
`addMEDIAN()` (*pyAgrum.BayesNet* method), 66  
`addMIN()` (*pyAgrum.BayesNet* method), 66  
`addNode()` (*pyAgrum.CliqueGraph* method), 15  
`addNode()` (*pyAgrum.DAG* method), 8  
`addNode()` (*pyAgrum.DiGraph* method), 5  
`addNode()` (*pyAgrum.MixedGraph* method), 20  
`addNode()` (*pyAgrum.UndiGraph* method), 12  
`addNodes()` (*pyAgrum.CliqueGraph* method), 15  
`addNodes()` (*pyAgrum.DAG* method), 8  
`addNodes()` (*pyAgrum.DiGraph* method), 5  
`addNodes()` (*pyAgrum.MixedGraph* method), 20  
`addNodes()` (*pyAgrum.UndiGraph* method), 12  
`addNodeWithId()` (*pyAgrum.CliqueGraph* method), 15  
`addNodeWithId()` (*pyAgrum.DAG* method), 8  
`addNodeWithId()` (*pyAgrum.DiGraph* method), 5  
`addNodeWithId()` (*pyAgrum.MixedGraph* method), 20  
`addNodeWithId()` (*pyAgrum.UndiGraph* method), 12  
`addNoForgettingAssumption()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 201  
`addNoisyAND()` (*pyAgrum.BayesNet* method), 66  
`addNoisyOR()` (*pyAgrum.BayesNet* method), 66  
`addNoisyORCompound()` (*pyAgrum.BayesNet* method), 67  
`addNoisyORNet()` (*pyAgrum.BayesNet* method), 67  
`addOR()` (*pyAgrum.BayesNet* method), 67  
`addPossibleEdge()` (*pyAgrum.BNLearner* method), 182  
`addStructureListener()` (*pyAgrum.BayesNet* method), 68  
`addStructureListener()` (*pyAgrum.BayesNetFragment* method), 90  
`addStructureListener()` (*pyAgrum.InfluenceDiagram* method), 193  
`addStructureListener()` (*pyAgrum.MarkovNet* method), 217  
`addSUM()` (*pyAgrum.BayesNet* method), 68  
`addTarget()` (*pyAgrum.GibbsSampling* method), 127  
`addTarget()` (*pyAgrum.ImportanceSampling* method), 148  
`addTarget()` (*pyAgrum.LazyPropagation* method), 99  
`addTarget()` (*pyAgrum.LoopyBeliefPropagation* method), 120  
`addTarget()` (*pyAgrum.LoopyGibbsSampling* method), 155  
`addTarget()` (*pyAgrum.LoopyImportanceSampling* method), 176  
`addTarget()` (*pyAgrum.LoopyMonteCarloSampling* method), 162  
`addTarget()` (*pyAgrum.LoopyWeightedSampling* method), 169  
`addTarget()` (*pyAgrum.MonteCarloSampling* method), 134  
`addTarget()` (*pyAgrum.ShaferShenoyInference* method), 107  
`addTarget()` (*pyAgrum.ShaferShenoyMNIInference* method), 224  
`addTarget()` (*pyAgrum.VariableElimination* method), 114  
`addTarget()` (*pyAgrum.WeightedSampling* method), 141  
`addTick()` (*pyAgrum.DiscretizedVariable* method), 32  
`addToClique()` (*pyAgrum.CliqueGraph* method), 15  
`addUtilityNode()` (*pyAgrum.InfluenceDiagram* method), 193  
`addValue()` (*pyAgrum.IntegerVariable* method), 35  
`addValue()` (*pyAgrum.NumericalDiscreteVariable* method), 42  
`addVariable()` (*pyAgrum.CredalNet* method), 204  
`addVariables()` (*pyAgrum.BayesNet* method), 68  
`addVariables()` (*pyAgrum.BayesNetFragment* method), 90  
`addVariables()` (*pyAgrum.InfluenceDiagram* method), 193  
`addVariables()` (*pyAgrum.MarkovNet* method), 218  
`addVarsFromModel()` (*pyAgrum.Instantiation* method), 47  
`addWeightedArc()` (*pyAgrum.BayesNet* method), 68  
`adjacents()` (*pyAgrum.MixedGraph* method), 20  
`aggType` (*pyAgrum.PRMEexplorer* property), 230  
`ancestors()` (*pyAgrum.BayesNet* method), 68  
`ancestors()` (*pyAgrum.BayesNetFragment* method), 90  
`ancestors()` (*pyAgrum.InfluenceDiagram* method), 193  
`animApproximationScheme()` (in module *pyAgrum.lib.notebook*), 268  
`approximatedBinarization()` (*pyAgrum.CredalNet* method), 204  
`Arc` (class in *pyAgrum*), 3  
`arcs()` (*pyAgrum.BayesNet* method), 69  
`arcs()` (*pyAgrum.BayesNetFragment* method), 90  
`arcs()` (*pyAgrum.causal.CausalModel* method), 236  
`arcs()` (*pyAgrum.DAG* method), 9



arcs() (*pyAgrum.DiGraph* method), 5  
 arcs() (*pyAgrum.EssentialGraph* method), 86  
 arcs() (*pyAgrum.InfluenceDiagram* method), 193  
 arcs() (*pyAgrum.MarkovBlanket* method), 88  
 arcs() (*pyAgrum.MixedGraph* method), 21  
 argmax() (*pyAgrum.Potential* method), 54  
 argmin() (*pyAgrum.Potential* method), 54  
 args (*pyAgrum.ArgumentError* attribute), 290  
 args (*pyAgrum.causal.HedgeException* attribute), 252  
 args (*pyAgrum.causal.UnidentifiableException* attribute), 252  
 args (*pyAgrum.CPTErrors* attribute), 292  
 args (*pyAgrum.DatabaseError* attribute), 292  
 args (*pyAgrum.DefaultInLabel* attribute), 287  
 args (*pyAgrum.DuplicateElement* attribute), 287  
 args (*pyAgrum.DuplicateLabel* attribute), 287  
 args (*pyAgrum.FatalError* attribute), 287  
 args (*pyAgrum.FormatNotFound* attribute), 287  
 args (*pyAgrum.GraphError* attribute), 288  
 args (*pyAgrum.GumException* attribute), 286  
 args (*pyAgrum.InvalidArc* attribute), 288  
 args (*pyAgrum.InvalidArgument* attribute), 288  
 args (*pyAgrum.InvalidArgumentsNumber* attribute), 288  
 args (*pyAgrum.InvalidDirectedCycle* attribute), 288  
 args (*pyAgrum.InvalidEdge* attribute), 289  
 args (*pyAgrum.InvalidNode* attribute), 289  
 args (*pyAgrum.IOError* attribute), 288  
 args (*pyAgrum.NoChild* attribute), 289  
 args (*pyAgrum.NoNeighbour* attribute), 289  
 args (*pyAgrum.NoParent* attribute), 289  
 args (*pyAgrum.NotFound* attribute), 289  
 args (*pyAgrum.NullElement* attribute), 290  
 args (*pyAgrum.OperationNotAllowed* attribute), 290  
 args (*pyAgrum.OutOfBounds* attribute), 290  
 args (*pyAgrum.SizeError* attribute), 290  
 args (*pyAgrum.SyntaxError* attribute), 291  
 args (*pyAgrum.UndefinedElement* attribute), 291  
 args (*pyAgrum.UndefinedIteratorKey* attribute), 291  
 args (*pyAgrum.UndefinedIteratorValue* attribute), 291  
 args (*pyAgrum.UnknownLabelInDatabase* attribute), 291  
 ArgumentError, 290  
 ASTBinaryOp (*class in pyAgrum.causal*), 242  
 ASTdiv (*class in pyAgrum.causal*), 246  
 ASTjointProba (*class in pyAgrum.causal*), 250  
 ASTminus (*class in pyAgrum.causal*), 245  
 ASTmult (*class in pyAgrum.causal*), 247  
 ASTplus (*class in pyAgrum.causal*), 244  
 ASTposteriorProba (*class in pyAgrum.causal*), 251  
 ASTsum (*class in pyAgrum.causal*), 249  
 ASTtree (*class in pyAgrum.causal*), 241  
 audit() (*pyAgrum.skbn.BNDiscretizer* method), 259  
 availableBNExts() (*in module pyAgrum*), 281  
 availableIDExts() (*in module pyAgrum*), 283  
 availableMNEExts() (*in module pyAgrum*), 282

## B

backDoor() (*pyAgrum.causal.CausalModel* method), 236  
 backdoor\_generator() (*in module pyAgrum.causal*), 240  
 BayesNet (*class in pyAgrum*), 64  
 BayesNetFragment (*class in pyAgrum*), 90  
 beginTopologyTransformation() (*pyAgrum.BayesNet* method), 69  
 beginTopologyTransformation() (*pyAgrum.MarkovNet* method), 218  
 belongs() (*pyAgrum.RangeVariable* method), 39  
 binaryJoinTree() (*pyAgrum.JunctionTreeGenerator* method), 85  
 bn (*pyAgrum.causal.ASTposteriorProba* property), 251  
 BN() (*pyAgrum.GibbsSampling* method), 126  
 BN() (*pyAgrum.ImportanceSampling* method), 147  
 BN() (*pyAgrum.LazyPropagation* method), 98  
 BN() (*pyAgrum.LoopyBeliefPropagation* method), 120  
 BN() (*pyAgrum.LoopyGibbsSampling* method), 154  
 BN() (*pyAgrum.LoopyImportanceSampling* method), 175  
 BN() (*pyAgrum.LoopyMonteCarloSampling* method), 161  
 BN() (*pyAgrum.LoopyWeightedSampling* method), 168  
 BN() (*pyAgrum.MonteCarloSampling* method), 134  
 BN() (*pyAgrum.ShaferShenoyInference* method), 105  
 BN() (*pyAgrum.VariableElimination* method), 113  
 BN() (*pyAgrum.WeightedSampling* method), 140  
 BNClassifier (*class in pyAgrum.skbn*), 254  
 BNDatabaseGenerator (*class in pyAgrum*), 78  
 BNDiscretizer (*class in pyAgrum.skbn*), 258  
 BNLearner (*class in pyAgrum*), 182  
 bnToCredal() (*pyAgrum.CredalNet* method), 204  
 burnIn() (*pyAgrum.GibbsBNdistance* method), 81  
 burnIn() (*pyAgrum.GibbsSampling* method), 127  
 burnIn() (*pyAgrum.LoopyGibbsSampling* method), 155

## C

causalBN() (*pyAgrum.causal.CausalModel* method), 236  
 CausalFormula (*class in pyAgrum.causal*), 238  
 causalImpact() (*in module pyAgrum.causal*), 239  
 CausalModel (*class in pyAgrum.causal*), 236  
 chanceNodeSize() (*pyAgrum.InfluenceDiagram* method), 194  
 changeLabel() (*pyAgrum.LabelizedVariable* method), 28  
 changePotential() (*pyAgrum.BayesNet* method), 69  
 changeValue() (*pyAgrum.IntegerVariable* method), 36  
 changeValue() (*pyAgrum.NumericalDiscreteVariable* method), 42  
 changeVariableLabel() (*pyAgrum.BayesNet* method), 69  
 changeVariableLabel() (*pyAgrum.MarkovNet* method), 218

`changeVariableName()` (*pyAgrum.BayesNet method*), 69  
`changeVariableName()` (*pyAgrum.InfluenceDiagram method*), 194  
`changeVariableName()` (*pyAgrum.MarkovNet method*), 218  
`check()` (*pyAgrum.BayesNet method*), 69  
`check()` (*pyAgrum.BayesNetFragment method*), 90  
`checkConsistency()` (*pyAgrum.BayesNetFragment method*), 91  
`chgEvidence()` (*pyAgrum.GibbsSampling method*), 127  
`chgEvidence()` (*pyAgrum.ImportanceSampling method*), 148  
`chgEvidence()` (*pyAgrum.LazyPropagation method*), 99  
`chgEvidence()` (*pyAgrum.LoopyBeliefPropagation method*), 121  
`chgEvidence()` (*pyAgrum.LoopyGibbsSampling method*), 155  
`chgEvidence()` (*pyAgrum.LoopyImportanceSampling method*), 176  
`chgEvidence()` (*pyAgrum.LoopyMonteCarloSampling method*), 162  
`chgEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 169  
`chgEvidence()` (*pyAgrum.MonteCarloSampling method*), 135  
`chgEvidence()` (*pyAgrum.ShaferShenoyInference method*), 107  
`chgEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 201  
`chgEvidence()` (*pyAgrum.ShaferShenoyMNIInference method*), 224  
`chgEvidence()` (*pyAgrum.VariableElimination method*), 114  
`chgEvidence()` (*pyAgrum.WeightedSampling method*), 141  
`chgVal()` (*pyAgrum.Instantiation method*), 47  
`chi2()` (*pyAgrum.BN Learner method*), 182  
`children()` (*pyAgrum.BayesNet method*), 69  
`children()` (*pyAgrum.BayesNetFragment method*), 91  
`children()` (*pyAgrum.causal.CausalModel method*), 236  
`children()` (*pyAgrum.DAG method*), 9  
`children()` (*pyAgrum.DiGraph method*), 5  
`children()` (*pyAgrum.EssentialGraph method*), 86  
`children()` (*pyAgrum.InfluenceDiagram method*), 194  
`children()` (*pyAgrum.MarkovBlanket method*), 88  
`children()` (*pyAgrum.MixedGraph method*), 21  
`classAggregates()` (*pyAgrum.PRMexplorer method*), 230  
`classAttributes()` (*pyAgrum.PRMexplorer method*), 230  
`classDag()` (*pyAgrum.PRMexplorer method*), 231  
`classes()` (*pyAgrum.PRMexplorer method*), 231  
`classImplements()` (*pyAgrum.PRMexplorer method*), 231  
`classParameters()` (*pyAgrum.PRMexplorer method*), 231  
`classReferences()` (*pyAgrum.PRMexplorer method*), 231  
`classSlotChains()` (*pyAgrum.PRMexplorer method*), 231  
`clear()` (*pyAgrum.BayesNet method*), 70  
`clear()` (*pyAgrum.CliqueGraph method*), 16  
`clear()` (*pyAgrum.DAG method*), 9  
`clear()` (*pyAgrum.DiGraph method*), 6  
`clear()` (*pyAgrum.InfluenceDiagram method*), 194  
`clear()` (*pyAgrum.Instantiation method*), 48  
`clear()` (*pyAgrum.MarkovNet method*), 218  
`clear()` (*pyAgrum.MixedGraph method*), 21  
`clear()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 201  
`clear()` (*pyAgrum.skbn.BNDiscretizer method*), 259  
`clear()` (*pyAgrum.UndiGraph method*), 12  
`clearEdges()` (*pyAgrum.CliqueGraph method*), 16  
`clique()` (*pyAgrum.CliqueGraph method*), 16  
`CliqueGraph` (class in *pyAgrum*), 15  
`closestIndex()` (*pyAgrum.NumericalDiscreteVariable method*), 43  
`closestLabel()` (*pyAgrum.NumericalDiscreteVariable method*), 43  
`cm` (*pyAgrum.causal.CausalFormula property*), 238  
`CN()` (*pyAgrum.CNLoopyPropagation method*), 212  
`CN()` (*pyAgrum.CNMonteCarloSampling method*), 209  
`CNLoopyPropagation` (class in *pyAgrum*), 212  
`CNMonteCarloSampling` (class in *pyAgrum*), 209  
`col()` (*pyAgrum.SyntaxError method*), 291  
`completeInstantiation()` (*pyAgrum.BayesNet method*), 70  
`completeInstantiation()` (*pyAgrum.BayesNetFragment method*), 91  
`completeInstantiation()` (*pyAgrum.InfluenceDiagram method*), 194  
`completeInstantiation()` (*pyAgrum.MarkovNet method*), 218  
`compute()` (*pyAgrum.ExactBNdistance method*), 80  
`compute()` (*pyAgrum.GibbsBNdistance method*), 81  
`computeBinaryCPTMinMax()` (*pyAgrum.CredalNet method*), 205  
`configuration()` (in module *pyAgrum.lib.notebook*), 269  
`connectedComponents()` (*pyAgrum.BayesNet method*), 70  
`connectedComponents()` (*pyAgrum.BayesNetFragment method*), 91  
`connectedComponents()` (*pyAgrum.CliqueGraph method*), 16  
`connectedComponents()` (*pyAgrum.DAG method*), 9  
`connectedComponents()` (*pyAgrum.DiGraph*

- method), 6
- connectedComponents() (pyAgrum.EssentialGraph method), 86
- connectedComponents() (pyAgrum.InfluenceDiagram method), 194
- connectedComponents() (pyAgrum.MarkovBlanket method), 88
- connectedComponents() (pyAgrum.MarkovNet method), 219
- connectedComponents() (pyAgrum.MixedGraph method), 21
- connectedComponents() (pyAgrum.UndiGraph method), 12
- container() (pyAgrum.CliqueGraph method), 16
- containerPath() (pyAgrum.CliqueGraph method), 16
- contains() (pyAgrum.Instantiation method), 48
- contains() (pyAgrum.Potential method), 54
- continueApproximationScheme() (pyAgrum.GibbsBNdistance method), 81
- copy() (pyAgrum.causal.ASTBinaryOp method), 242
- copy() (pyAgrum.causal.ASTdiv method), 246
- copy() (pyAgrum.causal.ASTjointProba method), 250
- copy() (pyAgrum.causal.ASTminus method), 245
- copy() (pyAgrum.causal.ASTMult method), 247
- copy() (pyAgrum.causal.ASTplus method), 244
- copy() (pyAgrum.causal.ASTposteriorProba method), 251
- copy() (pyAgrum.causal.ASTsum method), 249
- copy() (pyAgrum.causal.ASTtree method), 241
- copy() (pyAgrum.causal.CausalFormula method), 238
- correctedMutualInformation() (pyAgrum.BN Learner method), 183
- cpf() (pyAgrum.PRMEexplorer method), 232
- cpt() (pyAgrum.BayesNet method), 70
- cpt() (pyAgrum.BayesNetFragment method), 91
- cpt() (pyAgrum.InfluenceDiagram method), 194
- CPTError, 292
- createVariable() (pyAgrum.skbn.BNDiscretizer method), 259
- CredalNet (class in pyAgrum), 204
- credalNet\_currentCpt() (pyAgrum.CredalNet method), 205
- credalNet\_srcCpt() (pyAgrum.CredalNet method), 205
- current\_bn() (pyAgrum.CredalNet method), 205
- currentNodeType() (pyAgrum.CredalNet method), 205
- currentPosterior() (pyAgrum.GibbsSampling method), 128
- currentPosterior() (pyAgrum.ImportanceSampling method), 148
- currentPosterior() (pyAgrum.LoopyGibbsSampling method), 155
- currentPosterior() (pyAgrum.LoopyImportanceSampling method), 176
- currentPosterior() (pyAgrum.LoopyMonteCarloSampling method), 163
- currentPosterior() (pyAgrum.LoopyWeightedSampling method), 169
- currentPosterior() (pyAgrum.MonteCarloSampling method), 135
- currentPosterior() (pyAgrum.WeightedSampling method), 142
- currentTime() (pyAgrum.BN Learner method), 183
- currentTime() (pyAgrum.CNLoopyPropagation method), 213
- currentTime() (pyAgrum.CNMonteCarloSampling method), 209
- currentTime() (pyAgrum.GibbsBNdistance method), 81
- currentTime() (pyAgrum.GibbsSampling method), 128
- currentTime() (pyAgrum.ImportanceSampling method), 149
- currentTime() (pyAgrum.LoopyBeliefPropagation method), 121
- currentTime() (pyAgrum.LoopyGibbsSampling method), 155
- currentTime() (pyAgrum.LoopyImportanceSampling method), 176
- currentTime() (pyAgrum.LoopyMonteCarloSampling method), 163
- currentTime() (pyAgrum.LoopyWeightedSampling method), 170
- currentTime() (pyAgrum.MonteCarloSampling method), 135
- currentTime() (pyAgrum.WeightedSampling method), 142
- ## D
- DAG (class in pyAgrum), 8
- dag() (pyAgrum.BayesNet method), 70
- dag() (pyAgrum.BayesNetFragment method), 91
- dag() (pyAgrum.InfluenceDiagram method), 195
- dag() (pyAgrum.MarkovBlanket method), 88
- database() (pyAgrum.BNDatabaseGenerator method), 78
- DatabaseError, 292
- databaseWeight() (pyAgrum.BN Learner method), 183
- dec() (pyAgrum.Instantiation method), 48
- decIn() (pyAgrum.Instantiation method), 48
- decisionNodeSize() (pyAgrum.InfluenceDiagram method), 195
- decisionOrder() (pyAgrum.InfluenceDiagram method), 195
- decisionOrderExists() (pyAgrum.InfluenceDiagram method), 195
- decNotVar() (pyAgrum.Instantiation method), 48
- decOut() (pyAgrum.Instantiation method), 48
- decVar() (pyAgrum.Instantiation method), 48



DefaultInLabel, 287  
descendants() (*pyAgrum.BayesNet* method), 70  
descendants() (*pyAgrum.BayesNetFragment* method), 92  
descendants() (*pyAgrum.InfluenceDiagram* method), 195  
description() (*pyAgrum.DiscreteVariable* method), 25  
description() (*pyAgrum.DiscretizedVariable* method), 32  
description() (*pyAgrum.IntegerVariable* method), 36  
description() (*pyAgrum.LabelizedVariable* method), 28  
description() (*pyAgrum.NumericalDiscreteVariable* method), 43  
description() (*pyAgrum.RangeVariable* method), 39  
diff() (*pyAgrum.PyAgrumConfiguration* method), 292  
DiGraph (class in *pyAgrum*), 5  
dim() (*pyAgrum.BayesNet* method), 70  
dim() (*pyAgrum.BayesNetFragment* method), 92  
dim() (*pyAgrum.MarkovNet* method), 219  
disableEpsilon() (*pyAgrum.GibbsBNdistance* method), 81  
disableMaxIter() (*pyAgrum.GibbsBNdistance* method), 81  
disableMaxTime() (*pyAgrum.GibbsBNdistance* method), 81  
disableMinEpsilonRate() (*pyAgrum.GibbsBNdistance* method), 81  
DiscreteVariable (class in *pyAgrum*), 25  
discretizationCAIM() (*pyAgrum.skbn.BNDiscretizer* method), 259  
discretizationElbowMethodRotation() (*pyAgrum.skbn.BNDiscretizer* method), 260  
discretizationMDLP() (*pyAgrum.skbn.BNDiscretizer* method), 260  
discretizationNML() (*pyAgrum.skbn.BNDiscretizer* method), 260  
DiscretizedVariable (class in *pyAgrum*), 31  
doCalculusWithObservation() (in module *pyAgrum.causal*), 240  
domain() (*pyAgrum.DiscreteVariable* method), 25  
domain() (*pyAgrum.DiscretizedVariable* method), 32  
domain() (*pyAgrum.IntegerVariable* method), 36  
domain() (*pyAgrum.LabelizedVariable* method), 29  
domain() (*pyAgrum.NumericalDiscreteVariable* method), 43  
domain() (*pyAgrum.RangeVariable* method), 39  
domainSize() (*pyAgrum.BN Learner* method), 183  
domainSize() (*pyAgrum.CredalNet* method), 205  
domainSize() (*pyAgrum.DiscreteVariable* method), 25  
domainSize() (*pyAgrum.DiscretizedVariable* method), 32  
domainSize() (*pyAgrum.Instantiation* method), 49  
domainSize() (*pyAgrum.IntegerVariable* method), 36

domainSize() (*pyAgrum.LabelizedVariable* method), 29  
domainSize() (*pyAgrum.NumericalDiscreteVariable* method), 43  
domainSize() (*pyAgrum.Potential* method), 54  
domainSize() (*pyAgrum.RangeVariable* method), 40  
draw() (*pyAgrum.Potential* method), 54  
drawSamples() (*pyAgrum.BNDatabaseGenerator* method), 78  
dSeparation() (*pyAgrum.DAG* method), 9  
DuplicateElement, 287  
DuplicateLabel, 287  
dynamicExpMax() (*pyAgrum.CNLoopyPropagation* method), 213  
dynamicExpMax() (*pyAgrum.CNMonteCarloSampling* method), 209  
dynamicExpMin() (*pyAgrum.CNLoopyPropagation* method), 213  
dynamicExpMin() (*pyAgrum.CNMonteCarloSampling* method), 209

## E

Edge (class in *pyAgrum*), 4  
edges() (*pyAgrum.CliqueGraph* method), 16  
edges() (*pyAgrum.EssentialGraph* method), 87  
edges() (*pyAgrum.MarkovNet* method), 219  
edges() (*pyAgrum.MixedGraph* method), 21  
edges() (*pyAgrum.UndiGraph* method), 13  
eliminationOrder() (*pyAgrum.JunctionTreeGenerator* method), 86  
empty() (*pyAgrum.BayesNet* method), 70  
empty() (*pyAgrum.BayesNetFragment* method), 92  
empty() (*pyAgrum.CliqueGraph* method), 17  
empty() (*pyAgrum.DAG* method), 9  
empty() (*pyAgrum.DiGraph* method), 6  
empty() (*pyAgrum.DiscreteVariable* method), 25  
empty() (*pyAgrum.DiscretizedVariable* method), 32  
empty() (*pyAgrum.InfluenceDiagram* method), 195  
empty() (*pyAgrum.Instantiation* method), 49  
empty() (*pyAgrum.IntegerVariable* method), 36  
empty() (*pyAgrum.LabelizedVariable* method), 29  
empty() (*pyAgrum.MarkovNet* method), 219  
empty() (*pyAgrum.MixedGraph* method), 21  
empty() (*pyAgrum.NumericalDiscreteVariable* method), 43  
empty() (*pyAgrum.Potential* method), 54  
empty() (*pyAgrum.RangeVariable* method), 40  
empty() (*pyAgrum.UndiGraph* method), 13  
emptyArcs() (*pyAgrum.DAG* method), 9  
emptyArcs() (*pyAgrum.DiGraph* method), 6  
emptyArcs() (*pyAgrum.MixedGraph* method), 21  
emptyEdges() (*pyAgrum.CliqueGraph* method), 17  
emptyEdges() (*pyAgrum.MixedGraph* method), 21  
emptyEdges() (*pyAgrum.UndiGraph* method), 13

[enableEpsilon\(\)](#) ([pyAgrum.GibbsBNdistance method](#)), 81  
[enableMaxIter\(\)](#) ([pyAgrum.GibbsBNdistance method](#)), 81  
[enableMaxTime\(\)](#) ([pyAgrum.GibbsBNdistance method](#)), 82  
[enableMinEpsilonRate\(\)](#) ([pyAgrum.GibbsBNdistance method](#)), 82  
[end\(\)](#) ([pyAgrum.Instantiation method](#)), 49  
[endTopologyTransformation\(\)](#) ([pyAgrum.BayesNet method](#)), 71  
[endTopologyTransformation\(\)](#) ([pyAgrum.MarkovNet method](#)), 219  
[entropy\(\)](#) ([pyAgrum.Potential method](#)), 54  
[epsilon\(\)](#) ([pyAgrum.BN Learner method](#)), 183  
[epsilon\(\)](#) ([pyAgrum.CN Loopy Propagation method](#)), 213  
[epsilon\(\)](#) ([pyAgrum.CN Monte Carlo Sampling method](#)), 210  
[epsilon\(\)](#) ([pyAgrum.GibbsBNdistance method](#)), 82  
[epsilon\(\)](#) ([pyAgrum.GibbsSampling method](#)), 128  
[epsilon\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 149  
[epsilon\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 121  
[epsilon\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 156  
[epsilon\(\)](#) ([pyAgrum.LoopyImportanceSampling method](#)), 176  
[epsilon\(\)](#) ([pyAgrum.LoopyMonteCarloSampling method](#)), 163  
[epsilon\(\)](#) ([pyAgrum.LoopyWeightedSampling method](#)), 170  
[epsilon\(\)](#) ([pyAgrum.MonteCarloSampling method](#)), 135  
[epsilon\(\)](#) ([pyAgrum.WeightedSampling method](#)), 142  
[epsilonMax\(\)](#) ([pyAgrum.CredalNet method](#)), 205  
[epsilonMean\(\)](#) ([pyAgrum.CredalNet method](#)), 206  
[epsilonMin\(\)](#) ([pyAgrum.CredalNet method](#)), 206  
[erase\(\)](#) ([pyAgrum.BayesNet method](#)), 71  
[erase\(\)](#) ([pyAgrum.InfluenceDiagram method](#)), 195  
[erase\(\)](#) ([pyAgrum.Instantiation method](#)), 49  
[erase\(\)](#) ([pyAgrum.MarkovNet method](#)), 219  
[eraseAllEvidence\(\)](#) ([pyAgrum.CN Loopy Propagation method](#)), 213  
[eraseAllEvidence\(\)](#) ([pyAgrum.GibbsSampling method](#)), 128  
[eraseAllEvidence\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 149  
[eraseAllEvidence\(\)](#) ([pyAgrum.LazyPropagation method](#)), 100  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 121  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 156  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyImportanceSampling method](#)), 177  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyMonteCarloSampling method](#)), 163  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyWeightedSampling method](#)), 170  
[eraseAllEvidence\(\)](#) ([pyAgrum.MonteCarloSampling method](#)), 135  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 107  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyLIMIDInference method](#)), 201  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 225  
[eraseAllEvidence\(\)](#) ([pyAgrum.VariableElimination method](#)), 114  
[eraseAllEvidence\(\)](#) ([pyAgrum.WeightedSampling method](#)), 142  
[eraseAllJointTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 100  
[eraseAllJointTargets\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 107  
[eraseAllJointTargets\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 225  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 100  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 107  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 225  
[eraseAllTargets\(\)](#) ([pyAgrum.GibbsSampling method](#)), 128  
[eraseAllTargets\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 149  
[eraseAllTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 100  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 121  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 156  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyImportanceSampling method](#)), 177  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyMonteCarloSampling method](#)), 163  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyWeightedSampling method](#)), 170

- `eraseAllTargets()` (*pyAgrum.MonteCarloSampling method*), 135
- `eraseAllTargets()` (*pyAgrum.ShaferShenoyInference method*), 107
- `eraseAllTargets()` (*pyAgrum.ShaferShenoyMNIInference method*), 225
- `eraseAllTargets()` (*pyAgrum.VariableElimination method*), 114
- `eraseAllTargets()` (*pyAgrum.WeightedSampling method*), 142
- `eraseArc()` (*pyAgrum.BayesNet method*), 71
- `eraseArc()` (*pyAgrum.DAG method*), 9
- `eraseArc()` (*pyAgrum.DiGraph method*), 6
- `eraseArc()` (*pyAgrum.InfluenceDiagram method*), 195
- `eraseArc()` (*pyAgrum.MixedGraph method*), 22
- `eraseCausalArc()` (*pyAgrum.causal.CausalModel method*), 237
- `eraseChildren()` (*pyAgrum.DAG method*), 10
- `eraseChildren()` (*pyAgrum.DiGraph method*), 6
- `eraseChildren()` (*pyAgrum.MixedGraph method*), 22
- `eraseEdge()` (*pyAgrum.CliqueGraph method*), 17
- `eraseEdge()` (*pyAgrum.MixedGraph method*), 22
- `eraseEdge()` (*pyAgrum.UndiGraph method*), 13
- `eraseEvidence()` (*pyAgrum.GibbsSampling method*), 128
- `eraseEvidence()` (*pyAgrum.ImportanceSampling method*), 149
- `eraseEvidence()` (*pyAgrum.LazyPropagation method*), 100
- `eraseEvidence()` (*pyAgrum.LoopyBeliefPropagation method*), 121
- `eraseEvidence()` (*pyAgrum.LoopyGibbsSampling method*), 156
- `eraseEvidence()` (*pyAgrum.LoopyImportanceSampling method*), 177
- `eraseEvidence()` (*pyAgrum.LoopyMonteCarloSampling method*), 163
- `eraseEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 170
- `eraseEvidence()` (*pyAgrum.MonteCarloSampling method*), 136
- `eraseEvidence()` (*pyAgrum.ShaferShenoyInference method*), 108
- `eraseEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 201
- `eraseEvidence()` (*pyAgrum.ShaferShenoyMNIInference method*), 225
- `eraseEvidence()` (*pyAgrum.VariableElimination method*), 115
- `eraseEvidence()` (*pyAgrum.WeightedSampling method*), 142
- `eraseFactor()` (*pyAgrum.MarkovNet method*), 219
- `eraseForbiddenArc()` (*pyAgrum.BNLearner method*), 183
- `eraseFromClique()` (*pyAgrum.CliqueGraph method*), 17
- `eraseJointTarget()` (*pyAgrum.LazyPropagation method*), 100
- `eraseJointTarget()` (*pyAgrum.ShaferShenoyInference method*), 108
- `eraseJointTarget()` (*pyAgrum.ShaferShenoyMNIInference method*), 225
- `eraseJointTarget()` (*pyAgrum.VariableElimination method*), 115
- `eraseLabels()` (*pyAgrum.LabelizedVariable method*), 29
- `eraseMandatoryArc()` (*pyAgrum.BNLearner method*), 183
- `eraseNeighbours()` (*pyAgrum.CliqueGraph method*), 17
- `eraseNeighbours()` (*pyAgrum.MixedGraph method*), 22
- `eraseNeighbours()` (*pyAgrum.UndiGraph method*), 13
- `eraseNode()` (*pyAgrum.CliqueGraph method*), 17
- `eraseNode()` (*pyAgrum.DAG method*), 10
- `eraseNode()` (*pyAgrum.DiGraph method*), 6
- `eraseNode()` (*pyAgrum.MixedGraph method*), 22
- `eraseNode()` (*pyAgrum.UndiGraph method*), 13
- `eraseParents()` (*pyAgrum.DAG method*), 10
- `eraseParents()` (*pyAgrum.DiGraph method*), 6
- `eraseParents()` (*pyAgrum.MixedGraph method*), 22
- `erasePossibleEdge()` (*pyAgrum.BNLearner method*), 183
- `eraseTarget()` (*pyAgrum.GibbsSampling method*), 128
- `eraseTarget()` (*pyAgrum.ImportanceSampling method*), 149
- `eraseTarget()` (*pyAgrum.LazyPropagation method*), 101
- `eraseTarget()` (*pyAgrum.LoopyBeliefPropagation method*), 122
- `eraseTarget()` (*pyAgrum.LoopyGibbsSampling method*), 156
- `eraseTarget()` (*pyAgrum.LoopyImportanceSampling method*), 177
- `eraseTarget()` (*pyAgrum.LoopyMonteCarloSampling method*), 163
- `eraseTarget()` (*pyAgrum.LoopyWeightedSampling method*), 170
- `eraseTarget()` (*pyAgrum.MonteCarloSampling method*), 136
- `eraseTarget()` (*pyAgrum.ShaferShenoyInference method*), 108

- `eraseTarget()` (`pyAgrum.ShaferShenoyMNIInference` method), 225  
`eraseTarget()` (`pyAgrum.VariableElimination` method), 115  
`eraseTarget()` (`pyAgrum.WeightedSampling` method), 143  
`eraseTicks()` (`pyAgrum.DiscretizedVariable` method), 32  
`eraseValue()` (`pyAgrum.IntegerVariable` method), 36  
`eraseValue()` (`pyAgrum.NumericalDiscreteVariable` method), 43  
`eraseValues()` (`pyAgrum.IntegerVariable` method), 36  
`eraseValues()` (`pyAgrum.NumericalDiscreteVariable` method), 43  
`errorCallStack()` (`pyAgrum.GumException` method), 286  
`errorContent()` (`pyAgrum.GumException` method), 286  
`errorType()` (`pyAgrum.GumException` method), 286  
`EssentialGraph` (class in `pyAgrum`), 86  
`eval()` (`pyAgrum.causal.ASTBinaryOp` method), 243  
`eval()` (`pyAgrum.causal.ASTdiv` method), 246  
`eval()` (`pyAgrum.causal.ASTjointProba` method), 250  
`eval()` (`pyAgrum.causal.ASTminus` method), 245  
`eval()` (`pyAgrum.causal.ASTmult` method), 247  
`eval()` (`pyAgrum.causal.ASTplus` method), 244  
`eval()` (`pyAgrum.causal.ASTposteriorProba` method), 251  
`eval()` (`pyAgrum.causal.ASTsum` method), 249  
`eval()` (`pyAgrum.causal.ASTtree` method), 242  
`eval()` (`pyAgrum.causal.CausalFormula` method), 239  
`evidenceImpact()` (`pyAgrum.GibbsSampling` method), 129  
`evidenceImpact()` (`pyAgrum.ImportanceSampling` method), 149  
`evidenceImpact()` (`pyAgrum.LazyPropagation` method), 101  
`evidenceImpact()` (`pyAgrum.LoopyBeliefPropagation` method), 122  
`evidenceImpact()` (`pyAgrum.LoopyGibbsSampling` method), 156  
`evidenceImpact()` (`pyAgrum.LoopyImportanceSampling` method), 177  
`evidenceImpact()` (`pyAgrum.LoopyMonteCarloSampling` method), 164  
`evidenceImpact()` (`pyAgrum.LoopyWeightedSampling` method), 170  
`evidenceImpact()` (`pyAgrum.MonteCarloSampling` method), 136  
`evidenceImpact()` (`pyAgrum.ShaferShenoyInference` method), 108  
`evidenceImpact()` (`pyAgrum.ShaferShenoyMNIInference` method), 226  
`evidenceImpact()` (`pyAgrum.VariableElimination` method), 115  
`evidenceImpact()` (`pyAgrum.WeightedSampling` method), 143  
`evidenceJointImpact()` (`pyAgrum.LazyPropagation` method), 101  
`evidenceJointImpact()` (`pyAgrum.ShaferShenoyInference` method), 108  
`evidenceJointImpact()` (`pyAgrum.ShaferShenoyMNIInference` method), 226  
`evidenceJointImpact()` (`pyAgrum.VariableElimination` method), 115  
`evidenceProbability()` (`pyAgrum.LazyPropagation` method), 101  
`evidenceProbability()` (`pyAgrum.ShaferShenoyInference` method), 109  
`evidenceProbability()` (`pyAgrum.ShaferShenoyMNIInference` method), 226  
`ExactBNdistance` (class in `pyAgrum`), 80  
`exists()` (`pyAgrum.BayesNet` method), 71  
`exists()` (`pyAgrum.BayesNetFragment` method), 92  
`exists()` (`pyAgrum.InfluenceDiagram` method), 195  
`exists()` (`pyAgrum.MarkovNet` method), 219  
`existsArc()` (`pyAgrum.BayesNet` method), 71  
`existsArc()` (`pyAgrum.BayesNetFragment` method), 92  
`existsArc()` (`pyAgrum.causal.CausalModel` method), 237  
`existsArc()` (`pyAgrum.DAG` method), 10  
`existsArc()` (`pyAgrum.DiGraph` method), 7  
`existsArc()` (`pyAgrum.InfluenceDiagram` method), 196  
`existsArc()` (`pyAgrum.MixedGraph` method), 22  
`existsEdge()` (`pyAgrum.CliqueGraph` method), 17  
`existsEdge()` (`pyAgrum.MarkovNet` method), 219  
`existsEdge()` (`pyAgrum.MixedGraph` method), 22  
`existsEdge()` (`pyAgrum.UndiGraph` method), 13  
`existsNode()` (`pyAgrum.CliqueGraph` method), 18  
`existsNode()` (`pyAgrum.DAG` method), 10  
`existsNode()` (`pyAgrum.DiGraph` method), 7  
`existsNode()` (`pyAgrum.MixedGraph` method), 23  
`existsNode()` (`pyAgrum.UndiGraph` method), 13  
`existsPathBetween()` (`pyAgrum.InfluenceDiagram` method), 196  
`export()` (in module `pyAgrum.lib.image`), 269  
`exportInference()` (in module `pyAgrum.lib.image`), 269  
`extract()` (`pyAgrum.Potential` method), 55
- ## F
- `factor()` (`pyAgrum.MarkovNet` method), 219  
`factors()` (`pyAgrum.MarkovNet` method), 220  
`family()` (`pyAgrum.BayesNet` method), 71



`family()` (*pyAgrum.BayesNetFragment* method), 92  
`family()` (*pyAgrum.InfluenceDiagram* method), 196  
`fastBN()` (*in module pyAgrum*), 280  
`fastID()` (*in module pyAgrum*), 281  
`fastMN()` (*in module pyAgrum*), 280  
`fastPrototype()` (*pyAgrum.BayesNet* static method), 71  
`fastPrototype()` (*pyAgrum.InfluenceDiagram* static method), 196  
`fastPrototype()` (*pyAgrum.MarkovNet* static method), 220  
`fastToLatex()` (*pyAgrum.causal.ASTBinaryOp* method), 243  
`fastToLatex()` (*pyAgrum.causal.ASTdiv* method), 246  
`fastToLatex()` (*pyAgrum.causal.ASTjointProba* method), 250  
`fastToLatex()` (*pyAgrum.causal.ASTminus* method), 245  
`fastToLatex()` (*pyAgrum.causal.ASTMult* method), 248  
`fastToLatex()` (*pyAgrum.causal.ASTplus* method), 244  
`fastToLatex()` (*pyAgrum.causal.ASTposteriorProba* method), 251  
`fastToLatex()` (*pyAgrum.causal.ASTsum* method), 249  
`fastToLatex()` (*pyAgrum.causal.ASTtree* method), 242  
`FatalError`, 287  
`filename()` (*pyAgrum.SyntaxError* method), 291  
`fillConstraint()` (*pyAgrum.CredalNet* method), 206  
`fillConstraints()` (*pyAgrum.CredalNet* method), 206  
`fillWith()` (*pyAgrum.Potential* method), 55  
`fillWithFunction()` (*pyAgrum.Potential* method), 55  
`findAll()` (*pyAgrum.Potential* method), 56  
`first()` (*pyAgrum.Arc* method), 3  
`first()` (*pyAgrum.Edge* method), 4  
`fit()` (*pyAgrum.skbn.BNClassifier* method), 256  
`fitParameters()` (*pyAgrum.BNLearner* method), 183  
`FormatNotFound`, 287  
`fromBN()` (*pyAgrum.MarkovNet* static method), 220  
`fromdict()` (*pyAgrum.Instantiation* method), 49  
`fromTrainedModel()` (*pyAgrum.skbn.BNClassifier* method), 256  
`frontDoor()` (*pyAgrum.causal.CausalModel* method), 237  
`frontdoor_generator()` (*in module pyAgrum.causal*), 241

## G

`G2()` (*pyAgrum.BNLearner* method), 182  
`generateCPT()` (*pyAgrum.BayesNet* method), 72  
`generateCPTs()` (*pyAgrum.BayesNet* method), 72

`generateCSV()` (*in module pyAgrum*), 279  
`generateFactor()` (*pyAgrum.MarkovNet* method), 220  
`generateFactors()` (*pyAgrum.MarkovNet* method), 220  
`generateSample()` (*in module pyAgrum*), 279  
`get()` (*pyAgrum.Potential* method), 56  
`get()` (*pyAgrum.PyAgrumConfiguration* method), 292  
`get_binaryCPT_max()` (*pyAgrum.CredalNet* method), 206  
`get_binaryCPT_min()` (*pyAgrum.CredalNet* method), 207  
`get_params()` (*pyAgrum.skbn.BNClassifier* method), 257  
`getalltheSystems()` (*pyAgrum.PRMexplorer* method), 233  
`getBN()` (*in module pyAgrum.lib.notebook*), 263  
`getCausalImpact()` (*in module pyAgrum.causal.notebook*), 253  
`getCausalModel()` (*in module pyAgrum.causal.notebook*), 253  
`getCN()` (*in module pyAgrum.lib.notebook*), 264  
`getDecisionGraph()` (*pyAgrum.InfluenceDiagram* method), 197  
`getDirectSubClass()` (*pyAgrum.PRMexplorer* method), 232  
`getDirectSubInterfaces()` (*pyAgrum.PRMexplorer* method), 232  
`getDirectSubTypes()` (*pyAgrum.PRMexplorer* method), 232  
`getDot()` (*in module pyAgrum.lib.notebook*), 268  
`getGraph()` (*in module pyAgrum.lib.notebook*), 268  
`getImplementations()` (*pyAgrum.PRMexplorer* method), 232  
`getInference()` (*in module pyAgrum.lib.notebook*), 265  
`getInfluenceDiagram()` (*in module pyAgrum.lib.notebook*), 263  
`getInformation()` (*in module pyAgrum.lib.explain*), 271  
`getJunctionTree()` (*in module pyAgrum.lib.notebook*), 266  
`getLabelMap()` (*pyAgrum.PRMexplorer* method), 232  
`getLabels()` (*pyAgrum.PRMexplorer* method), 233  
`getMN()` (*in module pyAgrum.lib.notebook*), 264  
`getNumberOfLogicalProcessors()` (*in module pyAgrum*), 286  
`getNumberOfThreads()` (*in module pyAgrum*), 286  
`getNumberOfThreads()` (*pyAgrum.BNLearner* method), 183  
`getNumberOfThreads()` (*pyAgrum.LazyPropagation* method), 101  
`getNumberOfThreads()` (*pyAgrum.ShaferShenoyInference* method), 109  
`getNumberOfThreads()` (*pyAgrum.ShaferShenoyMNIInference* method), 226



- [getNumberOfThreads\(\)](#) (*pyAgrum.VariableElimination method*), 116  
[getPosterior\(\)](#) (*in module pyAgrum*), 278  
[getPosterior\(\)](#) (*in module pyAgrum.lib.notebook*), 267  
[getPotential\(\)](#) (*in module pyAgrum.lib.notebook*), 267  
[getSuperClass\(\)](#) (*pyAgrum.PRMexplorer method*), 233  
[getSuperInterface\(\)](#) (*pyAgrum.PRMexplorer method*), 233  
[getSuperType\(\)](#) (*pyAgrum.PRMexplorer method*), 233  
[GibbsBNdistance](#) (*class in pyAgrum*), 80  
[GibbsSampling](#) (*class in pyAgrum*), 126  
[graph\(\)](#) (*pyAgrum.MarkovNet method*), 220  
[GraphError](#), 288  
[grep\(\)](#) (*pyAgrum.PyAgrumConfiguration method*), 293  
[GumException](#), 286
- ## H
- [H\(\)](#) (*pyAgrum.GibbsSampling method*), 126  
[H\(\)](#) (*pyAgrum.ImportanceSampling method*), 147  
[H\(\)](#) (*pyAgrum.LazyPropagation method*), 98  
[H\(\)](#) (*pyAgrum.LoopyBeliefPropagation method*), 120  
[H\(\)](#) (*pyAgrum.LoopyGibbsSampling method*), 154  
[H\(\)](#) (*pyAgrum.LoopyImportanceSampling method*), 175  
[H\(\)](#) (*pyAgrum.LoopyMonteCarloSampling method*), 161  
[H\(\)](#) (*pyAgrum.LoopyWeightedSampling method*), 168  
[H\(\)](#) (*pyAgrum.MonteCarloSampling method*), 134  
[H\(\)](#) (*pyAgrum.ShaferShenoyInference method*), 106  
[H\(\)](#) (*pyAgrum.ShaferShenoyMNIInference method*), 223  
[H\(\)](#) (*pyAgrum.VariableElimination method*), 113  
[H\(\)](#) (*pyAgrum.WeightedSampling method*), 141  
[hamming\(\)](#) (*pyAgrum.Instantiation method*), 49  
[hardEvidenceNodes\(\)](#) (*pyAgrum.GibbsSampling method*), 129  
[hardEvidenceNodes\(\)](#) (*pyAgrum.ImportanceSampling method*), 150  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LazyPropagation method*), 101  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LoopyBeliefPropagation method*), 122  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LoopyGibbsSampling method*), 156  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LoopyImportanceSampling method*), 177  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LoopyMonteCarloSampling method*), 164  
[hardEvidenceNodes\(\)](#) (*pyAgrum.LoopyWeightedSampling method*), 171  
[hardEvidenceNodes\(\)](#) (*pyAgrum.MonteCarloSampling method*), 136  
[hardEvidenceNodes\(\)](#) (*pyAgrum.ShaferShenoyInference method*), 109  
[hardEvidenceNodes\(\)](#) (*pyAgrum.ShaferShenoyLIMIDInference method*), 201  
[hardEvidenceNodes\(\)](#) (*pyAgrum.ShaferShenoyMNIInference method*), 226  
[hardEvidenceNodes\(\)](#) (*pyAgrum.VariableElimination method*), 116  
[hardEvidenceNodes\(\)](#) (*pyAgrum.WeightedSampling method*), 143  
[hasComputedBinaryCPTMinMax\(\)](#) (*pyAgrum.CredalNet method*), 207  
[hasDirectedPath\(\)](#) (*pyAgrum.DAG method*), 10  
[hasDirectedPath\(\)](#) (*pyAgrum.DiGraph method*), 7  
[hasDirectedPath\(\)](#) (*pyAgrum.MixedGraph method*), 23  
[hasEvidence\(\)](#) (*pyAgrum.GibbsSampling method*), 129  
[hasEvidence\(\)](#) (*pyAgrum.ImportanceSampling method*), 150  
[hasEvidence\(\)](#) (*pyAgrum.LazyPropagation method*), 102  
[hasEvidence\(\)](#) (*pyAgrum.LoopyBeliefPropagation method*), 122  
[hasEvidence\(\)](#) (*pyAgrum.LoopyGibbsSampling method*), 157  
[hasEvidence\(\)](#) (*pyAgrum.LoopyImportanceSampling method*), 177  
[hasEvidence\(\)](#) (*pyAgrum.LoopyMonteCarloSampling method*), 164  
[hasEvidence\(\)](#) (*pyAgrum.LoopyWeightedSampling method*), 171  
[hasEvidence\(\)](#) (*pyAgrum.MonteCarloSampling method*), 136  
[hasEvidence\(\)](#) (*pyAgrum.ShaferShenoyInference method*), 109  
[hasEvidence\(\)](#) (*pyAgrum.ShaferShenoyLIMIDInference method*), 201  
[hasEvidence\(\)](#) (*pyAgrum.ShaferShenoyMNIInference method*), 226  
[hasEvidence\(\)](#) (*pyAgrum.VariableElimination method*), 116  
[hasEvidence\(\)](#) (*pyAgrum.WeightedSampling method*), 143  
[hasHardEvidence\(\)](#) (*pyAgrum.GibbsSampling method*), 129  
[hasHardEvidence\(\)](#) (*pyAgrum.ImportanceSampling method*), 150  
[hasHardEvidence\(\)](#) (*pyAgrum.LazyPropagation method*), 102  
[hasHardEvidence\(\)](#) (*pyAgrum.LoopyBeliefPropagation method*), 122  
[hasHardEvidence\(\)](#) (*pyAgrum.LoopyGibbsSampling method*), 156  
[hasHardEvidence\(\)](#) (*pyAgrum.LoopyImportanceSampling method*), 177  
[hasHardEvidence\(\)](#) (*pyAgrum.LoopyMonteCarloSampling method*), 164  
[hasHardEvidence\(\)](#) (*pyAgrum.LoopyWeightedSampling method*), 171  
[hasHardEvidence\(\)](#) (*pyAgrum.MonteCarloSampling method*), 136  
[hasHardEvidence\(\)](#) (*pyAgrum.ShaferShenoyInference method*), 109  
[hasHardEvidence\(\)](#) (*pyAgrum.ShaferShenoyLIMIDInference method*), 201  
[hasHardEvidence\(\)](#) (*pyAgrum.ShaferShenoyMNIInference method*), 226  
[hasHardEvidence\(\)](#) (*pyAgrum.VariableElimination method*), 116  
[hasHardEvidence\(\)](#) (*pyAgrum.WeightedSampling method*), 143

`grum.LoopyBeliefPropagation` method), 122

`hasHardEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 157

`hasHardEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 178

`hasHardEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 164

`hasHardEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 171

`hasHardEvidence()` (`pyAgrum.MonteCarloSampling` method), 136

`hasHardEvidence()` (`pyAgrum.ShaferShenoyInference` method), 109

`hasHardEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 201

`hasHardEvidence()` (`pyAgrum.ShaferShenoyMNIInference` method), 226

`hasHardEvidence()` (`pyAgrum.VariableElimination` method), 116

`hasHardEvidence()` (`pyAgrum.WeightedSampling` method), 143

`hasMissingValues()` (`pyAgrum.BN Learner` method), 183

`hasNoForgettingAssumption()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 202

`hasRunningIntersection()` (`pyAgrum.CliqueGraph` method), 18

`hasSameStructure()` (`pyAgrum.BayesNet` method), 72

`hasSameStructure()` (`pyAgrum.BayesNetFragment` method), 92

`hasSameStructure()` (`pyAgrum.InfluenceDiagram` method), 197

`hasSameStructure()` (`pyAgrum.MarkovBlanket` method), 89

`hasSameStructure()` (`pyAgrum.MarkovNet` method), 220

`hasSoftEvidence()` (`pyAgrum.GibbsSampling` method), 129

`hasSoftEvidence()` (`pyAgrum.ImportanceSampling` method), 150

`hasSoftEvidence()` (`pyAgrum.LazyPropagation` method), 102

`hasSoftEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 122

`hasSoftEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 157

`hasSoftEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 178

`hasSoftEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 164

`hasSoftEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 171

`hasSoftEvidence()` (`pyAgrum.MonteCarloSampling` method), 137

`hasSoftEvidence()` (`pyAgrum.ShaferShenoyInference` method), 109

`hasSoftEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 202

`hasSoftEvidence()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`hasSoftEvidence()` (`pyAgrum.VariableElimination` method), 116

`hasSoftEvidence()` (`pyAgrum.WeightedSampling` method), 143

`hasUndirectedCycle()` (`pyAgrum.CliqueGraph` method), 18

`hasUndirectedCycle()` (`pyAgrum.MixedGraph` method), 23

`hasUndirectedCycle()` (`pyAgrum.UndiGraph` method), 14

`head()` (`pyAgrum.Arc` method), 3

`HedgeException` (class in `pyAgrum.causal`), 252

`history()` (`pyAgrum.BN Learner` method), 183

`history()` (`pyAgrum.CN LoopyPropagation` method), 213

`history()` (`pyAgrum.CN MonteCarloSampling` method), 210

`history()` (`pyAgrum.GibbsBN distance` method), 82

`history()` (`pyAgrum.GibbsSampling` method), 130

`history()` (`pyAgrum.ImportanceSampling` method), 150

`history()` (`pyAgrum.LoopyBeliefPropagation` method), 123

`history()` (`pyAgrum.LoopyGibbsSampling` method), 157

`history()` (`pyAgrum.LoopyImportanceSampling` method), 178

`history()` (`pyAgrum.LoopyMonteCarloSampling` method), 164

`history()` (`pyAgrum.LoopyWeightedSampling` method), 171

`history()` (`pyAgrum.MonteCarloSampling` method), 137

`history()` (`pyAgrum.WeightedSampling` method), 144

|

`I()` (`pyAgrum.LazyPropagation` method), 98

`I()` (`pyAgrum.ShaferShenoyInference` method), 106

`I()` (`pyAgrum.ShaferShenoyMNIInference` method), 223

identifyingIntervention() (in module *pyAgrum.causal*), 240  
 idFromName() (*pyAgrum.BayesNet* method), 72  
 idFromName() (*pyAgrum.BayesNetFragment* method), 92  
 idFromName() (*pyAgrum.BNLearner* method), 184  
 idFromName() (*pyAgrum.causal.CausalModel* method), 237  
 idFromName() (*pyAgrum.InfluenceDiagram* method), 197  
 idFromName() (*pyAgrum.MarkovNet* method), 221  
 idmLearning() (*pyAgrum.CredalNet* method), 207  
 ids() (*pyAgrum.BayesNet* method), 73  
 ids() (*pyAgrum.BayesNetFragment* method), 92  
 ids() (*pyAgrum.InfluenceDiagram* method), 197  
 ids() (*pyAgrum.MarkovNet* method), 221  
 ImportanceSampling (class in *pyAgrum*), 147  
 inc() (*pyAgrum.Instantiation* method), 50  
 incIn() (*pyAgrum.Instantiation* method), 50  
 incNotVar() (*pyAgrum.Instantiation* method), 50  
 incOut() (*pyAgrum.Instantiation* method), 50  
 incVar() (*pyAgrum.Instantiation* method), 50  
 independenceListForPairs() (in module *pyAgrum.lib.explain*), 270  
 index() (*pyAgrum.DiscreteVariable* method), 25  
 index() (*pyAgrum.DiscretizedVariable* method), 32  
 index() (*pyAgrum.IntegerVariable* method), 36  
 index() (*pyAgrum.LabeledVariable* method), 29  
 index() (*pyAgrum.NumericalDiscreteVariable* method), 43  
 index() (*pyAgrum.RangeVariable* method), 40  
 inferenceType() (*pyAgrum.CNLoopyPropagation* method), 213  
 InferenceType\_nodeToNeighbours (*pyAgrum.CNLoopyPropagation* attribute), 212  
 InferenceType\_ordered (*pyAgrum.CNLoopyPropagation* attribute), 212  
 InferenceType\_randomOrder (*pyAgrum.CNLoopyPropagation* attribute), 212  
 InfluenceDiagram (class in *pyAgrum*), 190  
 influenceDiagram() (*pyAgrum.ShaferShenoyLIMIDInference* method), 202  
 initApproximationScheme() (*pyAgrum.GibbsBNdistance* method), 82  
 initRandom() (in module *pyAgrum*), 285  
 inOverflow() (*pyAgrum.Instantiation* method), 49  
 insertEvidenceFile() (*pyAgrum.CNLoopyPropagation* method), 213  
 insertEvidenceFile() (*pyAgrum.CNMonteCarloSampling* method), 210  
 insertModalsFile() (*pyAgrum.CNLoopyPropagation* method), 213  
 insertModalsFile() (*pyAgrum.CNMonteCarloSampling* method), 210  
 installAscendants() (*pyAgrum.BayesNetFragment* method), 92  
 installCPT() (*pyAgrum.BayesNetFragment* method), 93  
 installMarginal() (*pyAgrum.BayesNetFragment* method), 93  
 installNode() (*pyAgrum.BayesNetFragment* method), 93  
 Instantiation (class in *pyAgrum*), 47  
 instantiation() (*pyAgrum.CredalNet* method), 207  
 integerDomain() (*pyAgrum.IntegerVariable* method), 36  
 IntegerVariable (class in *pyAgrum*), 35  
 interAttributes() (*pyAgrum.PRMexplorer* method), 233  
 interfaces() (*pyAgrum.PRMexplorer* method), 234  
 interReferences() (*pyAgrum.PRMexplorer* method), 234  
 intervalToCredal() (*pyAgrum.CredalNet* method), 207  
 intervalToCredalWithFiles() (*pyAgrum.CredalNet* method), 207  
 InvalidArc, 288  
 InvalidArgument, 288  
 InvalidArgumentsNumber, 288  
 InvalidDirectedCycle, 288  
 InvalidEdge, 289  
 InvalidNode, 289  
 inverse() (*pyAgrum.Potential* method), 56  
 IOError, 288  
 isAttribute() (*pyAgrum.PRMexplorer* method), 234  
 isChanceNode() (*pyAgrum.InfluenceDiagram* method), 197  
 isClass() (*pyAgrum.PRMexplorer* method), 234  
 isDecisionNode() (*pyAgrum.InfluenceDiagram* method), 197  
 isDrawnAtRandom() (*pyAgrum.GibbsBNdistance* method), 82  
 isDrawnAtRandom() (*pyAgrum.GibbsSampling* method), 130  
 isDrawnAtRandom() (*pyAgrum.LoopyGibbsSampling* method), 157  
 isEmpirical() (*pyAgrum.DiscretizedVariable* method), 33  
 isEnabledEpsilon() (*pyAgrum.GibbsBNdistance* method), 82  
 isEnabledMaxIter() (*pyAgrum.GibbsBNdistance* method), 82  
 isEnabledMaxTime() (*pyAgrum.GibbsBNdistance* method), 82  
 isEnabledMinEpsilonRate() (*pyAgrum.GibbsBNdistance* method), 82  
 isGumNumberOfThreadsOverridden() (*pyAgrum.BNLearner* method), 184  
 isGumNumberOfThreadsOverridden() (*pyAgrum.LazyPropagation* method), 102

`isGumNumberOfThreadsOverriden()` (`pyAgrum.ShaferShenoyInference` method), 109

`isGumNumberOfThreadsOverriden()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`isGumNumberOfThreadsOverriden()` (`pyAgrum.VariableElimination` method), 116

`isIndependent()` (`pyAgrum.BayesNet` method), 73

`isIndependent()` (`pyAgrum.BayesNetFragment` method), 93

`isIndependent()` (`pyAgrum.InfluenceDiagram` method), 197

`isIndependent()` (`pyAgrum.MarkovNet` method), 221

`isInstalledNode()` (`pyAgrum.BayesNetFragment` method), 93

`isInterface()` (`pyAgrum.PRMexplorer` method), 234

`isJoinTree()` (`pyAgrum.CliqueGraph` method), 18

`isJointTarget()` (`pyAgrum.LazyPropagation` method), 102

`isJointTarget()` (`pyAgrum.ShaferShenoyInference` method), 110

`isJointTarget()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`isJointTarget()` (`pyAgrum.VariableElimination` method), 116

`isLabel()` (`pyAgrum.LabelizedVariable` method), 29

`isMutable()` (`pyAgrum.Instantiation` method), 50

`isNonZeroMap()` (`pyAgrum.Potential` method), 56

`isOMP()` (in module `pyAgrum`), 286

`isSeparatelySpecified()` (`pyAgrum.CredalNet` method), 208

`isSolvable()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 202

`isTarget()` (`pyAgrum.GibbsSampling` method), 130

`isTarget()` (`pyAgrum.ImportanceSampling` method), 150

`isTarget()` (`pyAgrum.LazyPropagation` method), 102

`isTarget()` (`pyAgrum.LoopyBeliefPropagation` method), 123

`isTarget()` (`pyAgrum.LoopyGibbsSampling` method), 157

`isTarget()` (`pyAgrum.LoopyImportanceSampling` method), 178

`isTarget()` (`pyAgrum.LoopyMonteCarloSampling` method), 165

`isTarget()` (`pyAgrum.LoopyWeightedSampling` method), 171

`isTarget()` (`pyAgrum.MonteCarloSampling` method), 137

`isTarget()` (`pyAgrum.ShaferShenoyInference` method), 110

`isTarget()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`isTarget()` (`pyAgrum.VariableElimination` method), 117

`isTarget()` (`pyAgrum.WeightedSampling` method), 144

`isTick()` (`pyAgrum.DiscretizedVariable` method), 33

`isType()` (`pyAgrum.PRMexplorer` method), 234

`isUtilityNode()` (`pyAgrum.InfluenceDiagram` method), 197

`isValue()` (`pyAgrum.IntegerVariable` method), 37

`isValue()` (`pyAgrum.NumericalDiscreteVariable` method), 43

## J

`jointMutualInformation()` (`pyAgrum.LazyPropagation` method), 103

`jointMutualInformation()` (`pyAgrum.ShaferShenoyInference` method), 110

`jointMutualInformation()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`jointMutualInformation()` (`pyAgrum.VariableElimination` method), 117

`jointPosterior()` (`pyAgrum.LazyPropagation` method), 103

`jointPosterior()` (`pyAgrum.ShaferShenoyInference` method), 110

`jointPosterior()` (`pyAgrum.ShaferShenoyMNIInference` method), 228

`jointPosterior()` (`pyAgrum.VariableElimination` method), 117

`jointProbability()` (`pyAgrum.BayesNet` method), 73

`jointProbability()` (`pyAgrum.BayesNetFragment` method), 93

`joinTree()` (`pyAgrum.LazyPropagation` method), 103

`joinTree()` (`pyAgrum.ShaferShenoyInference` method), 110

`joinTree()` (`pyAgrum.ShaferShenoyMNIInference` method), 227

`jointTargets()` (`pyAgrum.LazyPropagation` method), 103

`jointTargets()` (`pyAgrum.ShaferShenoyInference` method), 111

`jointTargets()` (`pyAgrum.ShaferShenoyMNIInference` method), 228

`jointTargets()` (`pyAgrum.VariableElimination` method), 117

`junctionTree()` (`pyAgrum.JunctionTreeGenerator` method), 86

`junctionTree()` (`pyAgrum.LazyPropagation` method), 103

`junctionTree()` (`pyAgrum.ShaferShenoyInference` method), 111

`junctionTree()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 202



- junctionTree() (pyAgrum.ShaferShenoyMNIInference method), 228
- junctionTree() (pyAgrum.VariableElimination method), 117
- JunctionTreeGenerator (class in pyAgrum), 85
- ## K
- KL() (pyAgrum.Potential method), 53
- knw (pyAgrum.causal.ASTposteriorProba property), 251
- ## L
- label() (pyAgrum.DiscreteVariable method), 25
- label() (pyAgrum.DiscretizedVariable method), 33
- label() (pyAgrum.IntegerVariable method), 37
- label() (pyAgrum.LabelizedVariable method), 29
- label() (pyAgrum.NumericalDiscreteVariable method), 44
- label() (pyAgrum.RangeVariable method), 40
- LabelizedVariable (class in pyAgrum), 27
- labels() (pyAgrum.DiscreteVariable method), 26
- labels() (pyAgrum.DiscretizedVariable method), 33
- labels() (pyAgrum.IntegerVariable method), 37
- labels() (pyAgrum.LabelizedVariable method), 29
- labels() (pyAgrum.NumericalDiscreteVariable method), 44
- labels() (pyAgrum.RangeVariable method), 40
- lagrangeNormalization() (pyAgrum.CredalNet method), 208
- latentVariables() (pyAgrum.BNLearner method), 184
- latentVariablesIds() (pyAgrum.causal.CausalModel method), 237
- latexQuery() (pyAgrum.causal.CausalFormula method), 239
- LazyPropagation (class in pyAgrum), 98
- learnBN() (pyAgrum.BNLearner method), 184
- learnDAG() (pyAgrum.BNLearner method), 184
- learnEssentialGraph() (pyAgrum.BNLearner method), 184
- learnMixedStructure() (pyAgrum.BNLearner method), 184
- learnParameters() (pyAgrum.BNLearner method), 184
- line() (pyAgrum.SyntaxError method), 291
- load() (pyAgrum.PRMexplorer method), 234
- load() (pyAgrum.PyAgrumConfiguration method), 293
- loadBIF() (pyAgrum.BayesNet method), 73
- loadBIFXML() (pyAgrum.BayesNet method), 73
- loadBIFXML() (pyAgrum.InfluenceDiagram method), 198
- loadBN() (in module pyAgrum), 281
- loadDSL() (pyAgrum.BayesNet method), 73
- loadID() (in module pyAgrum), 283
- loadMN() (in module pyAgrum), 282
- loadNET() (pyAgrum.BayesNet method), 73
- loadO3PRM() (pyAgrum.BayesNet method), 74
- loadUAI() (pyAgrum.BayesNet method), 74
- loadUAI() (pyAgrum.MarkovNet method), 221
- log10DomainSize() (pyAgrum.BayesNet method), 74
- log10DomainSize() (pyAgrum.BayesNetFragment method), 94
- log10DomainSize() (pyAgrum.InfluenceDiagram method), 198
- log10DomainSize() (pyAgrum.MarkovNet method), 221
- log2() (pyAgrum.Potential method), 56
- log2JointProbability() (pyAgrum.BayesNet method), 74
- log2JointProbability() (pyAgrum.BayesNetFragment method), 94
- log2likelihood() (pyAgrum.BNDatabaseGenerator method), 78
- logLikelihood() (pyAgrum.BNLearner method), 185
- loopIn() (pyAgrum.Potential method), 56
- LoopyBeliefPropagation (class in pyAgrum), 120
- LoopyGibbsSampling (class in pyAgrum), 154
- LoopyImportanceSampling (class in pyAgrum), 175
- LoopyMonteCarloSampling (class in pyAgrum), 161
- LoopyWeightedSampling (class in pyAgrum), 168
- ## M
- makeInference() (pyAgrum.CNLoopyPropagation method), 214
- makeInference() (pyAgrum.CNMonteCarloSampling method), 210
- makeInference() (pyAgrum.GibbsSampling method), 130
- makeInference() (pyAgrum.ImportanceSampling method), 151
- makeInference() (pyAgrum.LazyPropagation method), 103
- makeInference() (pyAgrum.LoopyBeliefPropagation method), 123
- makeInference() (pyAgrum.LoopyGibbsSampling method), 158
- makeInference() (pyAgrum.LoopyImportanceSampling method), 178
- makeInference() (pyAgrum.LoopyMonteCarloSampling method), 165
- makeInference() (pyAgrum.LoopyWeightedSampling method), 172
- makeInference() (pyAgrum.MonteCarloSampling method), 137
- makeInference() (pyAgrum.ShaferShenoyInference method), 111
- makeInference() (pyAgrum.ShaferShenoyLIMIDInference method), 202

`makeInference()` (*pyAgrum.ShaferShenoyMNIInference method*), 228  
`makeInference()` (*pyAgrum.VariableElimination method*), 118  
`makeInference()` (*pyAgrum.WeightedSampling method*), 144  
`makeInference_()` (*pyAgrum.LoopyGibbsSampling method*), 158  
`makeInference_()` (*pyAgrum.LoopyImportanceSampling method*), 179  
`makeInference_()` (*pyAgrum.LoopyMonteCarloSampling method*), 165  
`makeInference_()` (*pyAgrum.LoopyWeightedSampling method*), 172  
`marginalMax()` (*pyAgrum.CNLoopyPropagation method*), 214  
`marginalMax()` (*pyAgrum.CNMonteCarloSampling method*), 210  
`marginalMin()` (*pyAgrum.CNLoopyPropagation method*), 214  
`marginalMin()` (*pyAgrum.CNMonteCarloSampling method*), 210  
`margMaxIn()` (*pyAgrum.Potential method*), 57  
`margMaxOut()` (*pyAgrum.Potential method*), 57  
`margMinIn()` (*pyAgrum.Potential method*), 57  
`margMinOut()` (*pyAgrum.Potential method*), 57  
`margProdIn()` (*pyAgrum.Potential method*), 57  
`margProdOut()` (*pyAgrum.Potential method*), 58  
`margSumIn()` (*pyAgrum.Potential method*), 58  
`margSumOut()` (*pyAgrum.Potential method*), 58  
`MarkovBlanket` (*class in pyAgrum*), 88  
`MarkovNet` (*class in pyAgrum*), 217  
`max()` (*pyAgrum.Potential method*), 58  
`maxIter()` (*pyAgrum.BN Learner method*), 185  
`maxIter()` (*pyAgrum.CNLoopyPropagation method*), 214  
`maxIter()` (*pyAgrum.CNMonteCarloSampling method*), 211  
`maxIter()` (*pyAgrum.GibbsBNdistance method*), 83  
`maxIter()` (*pyAgrum.GibbsSampling method*), 130  
`maxIter()` (*pyAgrum.ImportanceSampling method*), 151  
`maxIter()` (*pyAgrum.LoopyBeliefPropagation method*), 123  
`maxIter()` (*pyAgrum.LoopyGibbsSampling method*), 158  
`maxIter()` (*pyAgrum.LoopyImportanceSampling method*), 179  
`maxIter()` (*pyAgrum.LoopyMonteCarloSampling method*), 165  
`maxIter()` (*pyAgrum.LoopyWeightedSampling method*), 172  
`maxIter()` (*pyAgrum.MonteCarloSampling method*), 137  
`maxIter()` (*pyAgrum.WeightedSampling method*), 144  
`maxNonOne()` (*pyAgrum.Potential method*), 58  
`maxNonOneParam()` (*pyAgrum.BayesNet method*), 74  
`maxNonOneParam()` (*pyAgrum.BayesNetFragment method*), 94  
`maxNonOneParam()` (*pyAgrum.MarkovNet method*), 221  
`maxParam()` (*pyAgrum.BayesNet method*), 74  
`maxParam()` (*pyAgrum.BayesNetFragment method*), 94  
`maxParam()` (*pyAgrum.MarkovNet method*), 221  
`maxTime()` (*pyAgrum.BN Learner method*), 185  
`maxTime()` (*pyAgrum.CNLoopyPropagation method*), 214  
`maxTime()` (*pyAgrum.CNMonteCarloSampling method*), 211  
`maxTime()` (*pyAgrum.GibbsBNdistance method*), 83  
`maxTime()` (*pyAgrum.GibbsSampling method*), 130  
`maxTime()` (*pyAgrum.ImportanceSampling method*), 151  
`maxTime()` (*pyAgrum.LoopyBeliefPropagation method*), 123  
`maxTime()` (*pyAgrum.LoopyGibbsSampling method*), 158  
`maxTime()` (*pyAgrum.LoopyImportanceSampling method*), 179  
`maxTime()` (*pyAgrum.LoopyMonteCarloSampling method*), 165  
`maxTime()` (*pyAgrum.LoopyWeightedSampling method*), 172  
`maxTime()` (*pyAgrum.MonteCarloSampling method*), 137  
`maxTime()` (*pyAgrum.WeightedSampling method*), 144  
`maxVal()` (*pyAgrum.RangeVariable method*), 40  
`maxVarDomainSize()` (*pyAgrum.BayesNet method*), 74  
`maxVarDomainSize()` (*pyAgrum.BayesNetFragment method*), 94  
`maxVarDomainSize()` (*pyAgrum.MarkovNet method*), 221  
`meanVar()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 202  
`messageApproximationScheme()` (*pyAgrum.BN Learner method*), 185  
`messageApproximationScheme()` (*pyAgrum.CNLoopyPropagation method*), 214  
`messageApproximationScheme()` (*pyAgrum.CNMonteCarloSampling method*), 211  
`messageApproximationScheme()` (*pyAgrum.GibbsBNdistance method*), 83  
`messageApproximationScheme()` (*pyAgrum.GibbsSampling method*), 130  
`messageApproximationScheme()` (*pyAgrum.ImportanceSampling method*), 151  
`messageApproximationScheme()` (*pyAgrum.LoopyBeliefPropagation method*), 123  
`messageApproximationScheme()` (*pyAgrum.MonteCarloSampling method*), 137

- grum.LoopyGibbsSampling* method), 158
- `messageApproximationScheme()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `messageApproximationScheme()` (*pyAgrum.LoopyMonteCarloSampling* method), 165
- `messageApproximationScheme()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `messageApproximationScheme()` (*pyAgrum.MonteCarloSampling* method), 138
- `messageApproximationScheme()` (*pyAgrum.WeightedSampling* method), 144
- `MEU()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 201
- `min()` (*pyAgrum.Potential* method), 58
- `minEpsilonRate()` (*pyAgrum.BN Learner* method), 185
- `minEpsilonRate()` (*pyAgrum.CNLoopyPropagation* method), 214
- `minEpsilonRate()` (*pyAgrum.CNMonteCarloSampling* method), 211
- `minEpsilonRate()` (*pyAgrum.GibbsBNdistance* method), 83
- `minEpsilonRate()` (*pyAgrum.GibbsSampling* method), 130
- `minEpsilonRate()` (*pyAgrum.ImportanceSampling* method), 151
- `minEpsilonRate()` (*pyAgrum.LoopyBeliefPropagation* method), 123
- `minEpsilonRate()` (*pyAgrum.LoopyGibbsSampling* method), 158
- `minEpsilonRate()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `minEpsilonRate()` (*pyAgrum.LoopyMonteCarloSampling* method), 165
- `minEpsilonRate()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `minEpsilonRate()` (*pyAgrum.MonteCarloSampling* method), 138
- `minEpsilonRate()` (*pyAgrum.WeightedSampling* method), 144
- `minimalCondSet()` (*pyAgrum.BayesNet* method), 75
- `minimalCondSet()` (*pyAgrum.BayesNetFragment* method), 94
- `minimalCondSet()` (*pyAgrum.MarkovNet* method), 221
- `minNonZero()` (*pyAgrum.Potential* method), 59
- `minNonZeroParam()` (*pyAgrum.BayesNet* method), 75
- `minNonZeroParam()` (*pyAgrum.BayesNetFragment* method), 94
- `minNonZeroParam()` (*pyAgrum.MarkovNet* method), 221
- `minParam()` (*pyAgrum.BayesNet* method), 75
- `minParam()` (*pyAgrum.BayesNetFragment* method), 94
- `minParam()` (*pyAgrum.MarkovNet* method), 221
- `minVal()` (*pyAgrum.RangeVariable* method), 40
- `MixedGraph` (class in *pyAgrum*), 20
- `mixedGraph()` (*pyAgrum.EssentialGraph* method), 87
- `mixedOrientedPath()` (*pyAgrum.MixedGraph* method), 23
- `mixedUnorientedPath()` (*pyAgrum.MixedGraph* method), 23
- `MN()` (*pyAgrum.ShaferShenoyMNInference* method), 223
- module
- `pyAgrum.causal.notebook`, 253
- `MonteCarloSampling` (class in *pyAgrum*), 134
- `moralGraph()` (*pyAgrum.BayesNet* method), 75
- `moralGraph()` (*pyAgrum.BayesNetFragment* method), 94
- `moralGraph()` (*pyAgrum.DAG* method), 10
- `moralGraph()` (*pyAgrum.InfluenceDiagram* method), 198
- `moralizedAncestralGraph()` (*pyAgrum.BayesNet* method), 75
- `moralizedAncestralGraph()` (*pyAgrum.BayesNetFragment* method), 95
- `moralizedAncestralGraph()` (*pyAgrum.DAG* method), 11
- `moralizedAncestralGraph()` (*pyAgrum.InfluenceDiagram* method), 198
- `mutualInformation()` (*pyAgrum.BN Learner* method), 185
- ## N
- `name()` (*pyAgrum.DiscreteVariable* method), 26
- `name()` (*pyAgrum.DiscretizedVariable* method), 33
- `name()` (*pyAgrum.IntegerVariable* method), 37
- `name()` (*pyAgrum.LabelizedVariable* method), 29
- `name()` (*pyAgrum.NumericalDiscreteVariable* method), 44
- `name()` (*pyAgrum.RangeVariable* method), 40
- `nameFromId()` (*pyAgrum.BN Learner* method), 185
- `names` (*pyAgrum.Potential* property), 59
- `names()` (*pyAgrum.BayesNet* method), 75
- `names()` (*pyAgrum.BayesNetFragment* method), 95
- `names()` (*pyAgrum.BN Learner* method), 185
- `names()` (*pyAgrum.causal.CausalModel* method), 237
- `names()` (*pyAgrum.InfluenceDiagram* method), 198
- `names()` (*pyAgrum.MarkovNet* method), 221
- `nbCols()` (*pyAgrum.BN Learner* method), 185
- `nbrDim()` (*pyAgrum.Instantiation* method), 50
- `nbrDim()` (*pyAgrum.Potential* method), 59
- `nbrDrawnVar()` (*pyAgrum.GibbsBNdistance* method), 83
- `nbrDrawnVar()` (*pyAgrum.GibbsSampling* method), 131
- `nbrDrawnVar()` (*pyAgrum.LoopyGibbsSampling* method), 158

- `nbrEvidence()` (*pyAgrum.GibbsSampling* method), 131
- `nbrEvidence()` (*pyAgrum.ImportanceSampling* method), 151
- `nbrEvidence()` (*pyAgrum.LazyPropagation* method), 104
- `nbrEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 124
- `nbrEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 158
- `nbrEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `nbrEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 165
- `nbrEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `nbrEvidence()` (*pyAgrum.MonteCarloSampling* method), 138
- `nbrEvidence()` (*pyAgrum.ShaferShenoyInference* method), 111
- `nbrEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 202
- `nbrEvidence()` (*pyAgrum.ShaferShenoyMNIInference* method), 228
- `nbrEvidence()` (*pyAgrum.VariableElimination* method), 118
- `nbrEvidence()` (*pyAgrum.WeightedSampling* method), 145
- `nbrHardEvidence()` (*pyAgrum.GibbsSampling* method), 131
- `nbrHardEvidence()` (*pyAgrum.ImportanceSampling* method), 151
- `nbrHardEvidence()` (*pyAgrum.LazyPropagation* method), 104
- `nbrHardEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 124
- `nbrHardEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 158
- `nbrHardEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `nbrHardEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 166
- `nbrHardEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `nbrHardEvidence()` (*pyAgrum.MonteCarloSampling* method), 138
- `nbrHardEvidence()` (*pyAgrum.ShaferShenoyInference* method), 111
- `nbrHardEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 202
- `nbrHardEvidence()` (*pyAgrum.ShaferShenoyMNIInference* method), 228
- `nbrHardEvidence()` (*pyAgrum.VariableElimination* method), 118
- `nbrHardEvidence()` (*pyAgrum.WeightedSampling* method), 145
- `nbrIterations()` (*pyAgrum.BNLearner* method), 185
- `nbrIterations()` (*pyAgrum.CNLoopyPropagation* method), 214
- `nbrIterations()` (*pyAgrum.CNMonteCarloSampling* method), 211
- `nbrIterations()` (*pyAgrum.GibbsBNdistance* method), 83
- `nbrIterations()` (*pyAgrum.GibbsSampling* method), 131
- `nbrIterations()` (*pyAgrum.ImportanceSampling* method), 151
- `nbrIterations()` (*pyAgrum.LoopyBeliefPropagation* method), 124
- `nbrIterations()` (*pyAgrum.LoopyGibbsSampling* method), 159
- `nbrIterations()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `nbrIterations()` (*pyAgrum.LoopyMonteCarloSampling* method), 166
- `nbrIterations()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `nbrIterations()` (*pyAgrum.MonteCarloSampling* method), 138
- `nbrIterations()` (*pyAgrum.WeightedSampling* method), 145
- `nbrJointTargets()` (*pyAgrum.LazyPropagation* method), 104
- `nbrJointTargets()` (*pyAgrum.ShaferShenoyInference* method), 111
- `nbrJointTargets()` (*pyAgrum.ShaferShenoyMNIInference* method), 228
- `nbRows()` (*pyAgrum.BNLearner* method), 185
- `nbrSoftEvidence()` (*pyAgrum.GibbsSampling* method), 131
- `nbrSoftEvidence()` (*pyAgrum.ImportanceSampling* method), 152
- `nbrSoftEvidence()` (*pyAgrum.LazyPropagation* method), 104
- `nbrSoftEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 124
- `nbrSoftEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 159
- `nbrSoftEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 179
- `nbrSoftEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 166
- `nbrSoftEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 172
- `nbrSoftEvidence()` (*pyAgrum.MonteCarloSampling* method), 138
- `nbrSoftEvidence()` (*pyAgrum.WeightedSampling* method), 145



- grum.LoopyImportanceSampling* (method), 179
- nbrSoftEvidence()* (*pyAgrum.LoopyMonteCarloSampling* method), 166
- nbrSoftEvidence()* (*pyAgrum.LoopyWeightedSampling* method), 173
- nbrSoftEvidence()* (*pyAgrum.MonteCarloSampling* method), 138
- nbrSoftEvidence()* (*pyAgrum.ShaferShenoyInference* method), 111
- nbrSoftEvidence()* (*pyAgrum.ShaferShenoyLIMIDInference* method), 202
- nbrSoftEvidence()* (*pyAgrum.ShaferShenoyMNIInference* method), 229
- nbrSoftEvidence()* (*pyAgrum.VariableElimination* method), 118
- nbrSoftEvidence()* (*pyAgrum.WeightedSampling* method), 145
- nbrTargets()* (*pyAgrum.GibbsSampling* method), 131
- nbrTargets()* (*pyAgrum.ImportanceSampling* method), 152
- nbrTargets()* (*pyAgrum.LazyPropagation* method), 104
- nbrTargets()* (*pyAgrum.LoopyBeliefPropagation* method), 124
- nbrTargets()* (*pyAgrum.LoopyGibbsSampling* method), 159
- nbrTargets()* (*pyAgrum.LoopyImportanceSampling* method), 179
- nbrTargets()* (*pyAgrum.LoopyMonteCarloSampling* method), 166
- nbrTargets()* (*pyAgrum.LoopyWeightedSampling* method), 173
- nbrTargets()* (*pyAgrum.MonteCarloSampling* method), 138
- nbrTargets()* (*pyAgrum.ShaferShenoyInference* method), 111
- nbrTargets()* (*pyAgrum.ShaferShenoyMNIInference* method), 229
- nbrTargets()* (*pyAgrum.VariableElimination* method), 118
- nbrTargets()* (*pyAgrum.WeightedSampling* method), 145
- neighbours()* (*pyAgrum.CliqueGraph* method), 18
- neighbours()* (*pyAgrum.EssentialGraph* method), 87
- neighbours()* (*pyAgrum.MarkovNet* method), 221
- neighbours()* (*pyAgrum.MixedGraph* method), 23
- neighbours()* (*pyAgrum.UndiGraph* method), 14
- new\_abs()* (*pyAgrum.Potential* method), 59
- new\_log2()* (*pyAgrum.Potential* method), 59
- new\_sq()* (*pyAgrum.Potential* method), 59
- newFactory()* (*pyAgrum.Potential* method), 59
- NoChild, 289
- nodeId()* (*pyAgrum.BayesNet* method), 75
- nodeId()* (*pyAgrum.BayesNetFragment* method), 95
- nodeId()* (*pyAgrum.InfluenceDiagram* method), 198
- nodeId()* (*pyAgrum.MarkovNet* method), 222
- nodes()* (*pyAgrum.BayesNet* method), 76
- nodes()* (*pyAgrum.BayesNetFragment* method), 95
- nodes()* (*pyAgrum.causal.CausalModel* method), 237
- nodes()* (*pyAgrum.CliqueGraph* method), 18
- nodes()* (*pyAgrum.DAG* method), 11
- nodes()* (*pyAgrum.DiGraph* method), 7
- nodes()* (*pyAgrum.EssentialGraph* method), 87
- nodes()* (*pyAgrum.InfluenceDiagram* method), 198
- nodes()* (*pyAgrum.MarkovBlanket* method), 89
- nodes()* (*pyAgrum.MarkovNet* method), 222
- nodes()* (*pyAgrum.MixedGraph* method), 24
- nodes()* (*pyAgrum.UndiGraph* method), 14
- nodes2ConnectedComponent()* (*pyAgrum.CliqueGraph* method), 18
- nodes2ConnectedComponent()* (*pyAgrum.MixedGraph* method), 24
- nodes2ConnectedComponent()* (*pyAgrum.UndiGraph* method), 14
- nodeset()* (*pyAgrum.BayesNet* method), 76
- nodeset()* (*pyAgrum.BayesNetFragment* method), 95
- nodeset()* (*pyAgrum.InfluenceDiagram* method), 198
- nodeset()* (*pyAgrum.MarkovNet* method), 222
- nodeType()* (*pyAgrum.CredalNet* method), 208
- NodeType\_Credal* (*pyAgrum.CredalNet* attribute), 204
- NodeType\_Indic* (*pyAgrum.CredalNet* attribute), 204
- NodeType\_Precise* (*pyAgrum.CredalNet* attribute), 204
- NodeType\_Vacuous* (*pyAgrum.CredalNet* attribute), 204
- noising()* (*pyAgrum.Potential* method), 59
- NoNeighbour, 289
- NoParent, 289
- normalize()* (*pyAgrum.Potential* method), 59
- normalizeAsCPT()* (*pyAgrum.Potential* method), 59
- NotFound, 289
- NullElement, 290
- numerical()* (*pyAgrum.DiscreteVariable* method), 26
- numerical()* (*pyAgrum.DiscretizedVariable* method), 33
- numerical()* (*pyAgrum.IntegerVariable* method), 37
- numerical()* (*pyAgrum.LabelizedVariable* method), 30
- numerical()* (*pyAgrum.NumericalDiscreteVariable* method), 44
- numerical()* (*pyAgrum.RangeVariable* method), 40
- NumericalDiscreteVariable* (class in *pyAgrum*), 42
- numericalDomain()* (*pyAgrum.NumericalDiscreteVariable* method), 44

## O

*observationalBN()* (*pyAgrum.causal.CausalModel*

method), 238  
 op1 (*pyAgrum.causal.ASTBinaryOp* property), 243  
 op1 (*pyAgrum.causal.ASTdiv* property), 246  
 op1 (*pyAgrum.causal.ASTminus* property), 245  
 op1 (*pyAgrum.causal.ASTmult* property), 248  
 op1 (*pyAgrum.causal.ASTplus* property), 244  
 op2 (*pyAgrum.causal.ASTBinaryOp* property), 243  
 op2 (*pyAgrum.causal.ASTdiv* property), 247  
 op2 (*pyAgrum.causal.ASTminus* property), 245  
 op2 (*pyAgrum.causal.ASTmult* property), 248  
 op2 (*pyAgrum.causal.ASTplus* property), 244  
 OperationNotAllowed, 290  
 optimalDecision() (*pyAgrum.ShaferShenoyLIMIDInference* method), 202  
 other() (*pyAgrum.Arc* method), 3  
 other() (*pyAgrum.Edge* method), 4  
 OutOfBounds, 290

## P

parents() (*pyAgrum.BayesNet* method), 76  
 parents() (*pyAgrum.BayesNetFragment* method), 95  
 parents() (*pyAgrum.causal.CausalModel* method), 238  
 parents() (*pyAgrum.DAG* method), 11  
 parents() (*pyAgrum.DiGraph* method), 7  
 parents() (*pyAgrum.EssentialGraph* method), 87  
 parents() (*pyAgrum.InfluenceDiagram* method), 199  
 parents() (*pyAgrum.MarkovBlanket* method), 89  
 parents() (*pyAgrum.MixedGraph* method), 24  
 partialUndiGraph() (*pyAgrum.CliqueGraph* method), 18  
 partialUndiGraph() (*pyAgrum.MixedGraph* method), 24  
 partialUndiGraph() (*pyAgrum.UndiGraph* method), 14  
 periodSize() (*pyAgrum.BN Learner* method), 186  
 periodSize() (*pyAgrum.CN LoopyPropagation* method), 215  
 periodSize() (*pyAgrum.CN MonteCarloSampling* method), 211  
 periodSize() (*pyAgrum.GibbsBNdistance* method), 83  
 periodSize() (*pyAgrum.GibbsSampling* method), 131  
 periodSize() (*pyAgrum.ImportanceSampling* method), 152  
 periodSize() (*pyAgrum.LoopyBeliefPropagation* method), 124  
 periodSize() (*pyAgrum.LoopyGibbsSampling* method), 159  
 periodSize() (*pyAgrum.LoopyImportanceSampling* method), 180  
 periodSize() (*pyAgrum.LoopyMonteCarloSampling* method), 166  
 periodSize() (*pyAgrum.LoopyWeightedSampling* method), 173

periodSize() (*pyAgrum.MonteCarloSampling* method), 138  
 periodSize() (*pyAgrum.WeightedSampling* method), 145  
 pos() (*pyAgrum.Instantiation* method), 50  
 pos() (*pyAgrum.Potential* method), 60  
 posLabel() (*pyAgrum.LabelizedVariable* method), 30  
 posterior() (*pyAgrum.GibbsSampling* method), 131  
 posterior() (*pyAgrum.ImportanceSampling* method), 152  
 posterior() (*pyAgrum.LazyPropagation* method), 104  
 posterior() (*pyAgrum.LoopyBeliefPropagation* method), 124  
 posterior() (*pyAgrum.LoopyGibbsSampling* method), 159  
 posterior() (*pyAgrum.LoopyImportanceSampling* method), 180  
 posterior() (*pyAgrum.LoopyMonteCarloSampling* method), 166  
 posterior() (*pyAgrum.LoopyWeightedSampling* method), 173  
 posterior() (*pyAgrum.MonteCarloSampling* method), 138  
 posterior() (*pyAgrum.ShaferShenoyInference* method), 111  
 posterior() (*pyAgrum.ShaferShenoyLIMIDInference* method), 202  
 posterior() (*pyAgrum.ShaferShenoyMNIInference* method), 229  
 posterior() (*pyAgrum.VariableElimination* method), 118  
 posterior() (*pyAgrum.WeightedSampling* method), 145  
 posteriorUtility() (*pyAgrum.ShaferShenoyLIMIDInference* method), 202  
 Potential (class in *pyAgrum*), 53  
 predict() (*pyAgrum.skbn.BNClassifier* method), 257  
 predict\_proba() (*pyAgrum.skbn.BNClassifier* method), 257  
 preparedData() (*pyAgrum.skbn.BNClassifier* method), 257  
 PRMexplorer (class in *pyAgrum*), 230  
 product() (*pyAgrum.Potential* method), 60  
 property() (*pyAgrum.BayesNetFragment* method), 95  
 propertyWithDefault() (*pyAgrum.BayesNetFragment* method), 95  
 protectToLatex() (*pyAgrum.causal.ASTBinaryOp* method), 243  
 protectToLatex() (*pyAgrum.causal.ASTdiv* method), 247  
 protectToLatex() (*pyAgrum.causal.ASTjointProba* method), 250  
 protectToLatex() (*pyAgrum.causal.ASTminus* method), 246  
 protectToLatex() (*pyAgrum.causal.ASTmult* method), 248

- `protectToLatex()` (*pyAgrum.causal.ASTplus method*), 244  
`protectToLatex()` (*pyAgrum.causal.ASTposteriorProba method*), 251  
`protectToLatex()` (*pyAgrum.causal.ASTsum method*), 249  
`protectToLatex()` (*pyAgrum.causal.ASTtree method*), 242  
`pseudoCount()` (*pyAgrum.BN Learner method*), 186  
`putFirst()` (*pyAgrum.Potential method*), 60  
`pyAgrum.causal.notebook` module, 253  
`PyAgrumConfiguration` (class in *pyAgrum*), 292
- ## R
- `random()` (*pyAgrum.Potential method*), 60  
`randomCPT()` (*pyAgrum.Potential method*), 60  
`randomDistribution()` (in module *pyAgrum*), 286  
`randomDistribution()` (*pyAgrum.Potential method*), 60  
`randomProba()` (in module *pyAgrum*), 285  
`RangeVariable` (class in *pyAgrum*), 39  
`rawPseudoCount()` (*pyAgrum.BN Learner method*), 186  
`recordWeight()` (*pyAgrum.BN Learner method*), 186  
`reducedGraph()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 203  
`reducedLIMID()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 203  
`remainingBurnIn()` (*pyAgrum.GibbsBNdistance method*), 83  
`remove()` (*pyAgrum.Potential method*), 60  
`rend()` (*pyAgrum.Instantiation method*), 51  
`reorder()` (*pyAgrum.Instantiation method*), 51  
`reorganize()` (*pyAgrum.Potential method*), 61  
`reset()` (*pyAgrum.PyAgrumConfiguration method*), 293  
`reverseArc()` (*pyAgrum.BayesNet method*), 76  
`reversePartialOrder()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 203  
`root` (*pyAgrum.causal.CausalFormula property*), 239  
`run_hooks()` (*pyAgrum.PyAgrumConfiguration method*), 293
- ## S
- `samplesAt()` (*pyAgrum.BNDatabaseGenerator method*), 79  
`samplesLabelAt()` (*pyAgrum.BNDatabaseGenerator method*), 79  
`samplesNbCols()` (*pyAgrum.BNDatabaseGenerator method*), 79  
`samplesNbRows()` (*pyAgrum.BNDatabaseGenerator method*), 79  
`save()` (*pyAgrum.PyAgrumConfiguration method*), 293  
`saveBIF()` (*pyAgrum.BayesNet method*), 76  
`saveBIFXML()` (*pyAgrum.BayesNet method*), 76  
`saveBIFXML()` (*pyAgrum.InfluenceDiagram method*), 199  
`saveBN()` (in module *pyAgrum*), 282  
`saveBNsMinMax()` (*pyAgrum.CredalNet method*), 208  
`saveDSL()` (*pyAgrum.BayesNet method*), 76  
`saveID()` (in module *pyAgrum*), 284  
`saveInference()` (*pyAgrum.CNLoopyPropagation method*), 215  
`saveMN()` (in module *pyAgrum*), 283  
`saveNET()` (*pyAgrum.BayesNet method*), 76  
`saveO3PRM()` (*pyAgrum.BayesNet method*), 77  
`saveUAI()` (*pyAgrum.BayesNet method*), 77  
`saveUAI()` (*pyAgrum.MarkovNet method*), 222  
`scale()` (*pyAgrum.Potential method*), 61  
`score()` (*pyAgrum.BN Learner method*), 186  
`score()` (*pyAgrum.skbn.BNClassifier method*), 258  
`second()` (*pyAgrum.Arc method*), 4  
`second()` (*pyAgrum.Edge method*), 4  
`separator()` (*pyAgrum.CliqueGraph method*), 18  
`set()` (*pyAgrum.Potential method*), 61  
`set()` (*pyAgrum.PyAgrumConfiguration method*), 293  
`set_params()` (*pyAgrum.skbn.BNClassifier method*), 258  
`setAntiTopologicalVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 79  
`setBurnIn()` (*pyAgrum.GibbsBNdistance method*), 83  
`setBurnIn()` (*pyAgrum.GibbsSampling method*), 132  
`setBurnIn()` (*pyAgrum.LoopyGibbsSampling method*), 159  
`setClique()` (*pyAgrum.CliqueGraph method*), 19  
`setCPT()` (*pyAgrum.CredalNet method*), 208  
`setCPTs()` (*pyAgrum.CredalNet method*), 209  
`setDatabaseWeight()` (*pyAgrum.BN Learner method*), 186  
`setDescription()` (*pyAgrum.DiscreteVariable method*), 26  
`setDescription()` (*pyAgrum.DiscretizedVariable method*), 33  
`setDescription()` (*pyAgrum.IntegerVariable method*), 37  
`setDescription()` (*pyAgrum.LabelizedVariable method*), 30  
`setDescription()` (*pyAgrum.NumericalDiscreteVariable method*), 44  
`setDescription()` (*pyAgrum.RangeVariable method*), 41  
`setDiscretizationParameters()` (*pyAgrum.skbn.BNDiscretizer method*), 261  
`setDrawnAtRandom()` (*pyAgrum.GibbsBNdistance method*), 83  
`setDrawnAtRandom()` (*pyAgrum.GibbsSampling method*), 132

`setDrawnAtRandom()` (*pyAgrum.LoopyGibbsSampling method*), 159

`setEmpirical()` (*pyAgrum.DiscretizedVariable method*), 33

`setEpsilon()` (*pyAgrum.BN Learner method*), 186

`setEpsilon()` (*pyAgrum.CNLoopyPropagation method*), 215

`setEpsilon()` (*pyAgrum.CNMonteCarloSampling method*), 211

`setEpsilon()` (*pyAgrum.GibbsBNdistance method*), 84

`setEpsilon()` (*pyAgrum.GibbsSampling method*), 132

`setEpsilon()` (*pyAgrum.ImportanceSampling method*), 152

`setEpsilon()` (*pyAgrum.LoopyBeliefPropagation method*), 124

`setEpsilon()` (*pyAgrum.LoopyGibbsSampling method*), 159

`setEpsilon()` (*pyAgrum.LoopyImportanceSampling method*), 180

`setEpsilon()` (*pyAgrum.LoopyMonteCarloSampling method*), 166

`setEpsilon()` (*pyAgrum.LoopyWeightedSampling method*), 173

`setEpsilon()` (*pyAgrum.MonteCarloSampling method*), 139

`setEpsilon()` (*pyAgrum.WeightedSampling method*), 145

`setEvidence()` (*pyAgrum.GibbsSampling method*), 132

`setEvidence()` (*pyAgrum.ImportanceSampling method*), 152

`setEvidence()` (*pyAgrum.LazyPropagation method*), 104

`setEvidence()` (*pyAgrum.LoopyBeliefPropagation method*), 125

`setEvidence()` (*pyAgrum.LoopyGibbsSampling method*), 160

`setEvidence()` (*pyAgrum.LoopyImportanceSampling method*), 180

`setEvidence()` (*pyAgrum.LoopyMonteCarloSampling method*), 166

`setEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 173

`setEvidence()` (*pyAgrum.MonteCarloSampling method*), 139

`setEvidence()` (*pyAgrum.ShaferShenoyInference method*), 112

`setEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 203

`setEvidence()` (*pyAgrum.ShaferShenoyMNIInference method*), 229

`setEvidence()` (*pyAgrum.VariableElimination method*), 118

`setEvidence()` (*pyAgrum.WeightedSampling method*), 146

`setFirst()` (*pyAgrum.Instantiation method*), 51

`setFirstIn()` (*pyAgrum.Instantiation method*), 51

`setFirstNotVar()` (*pyAgrum.Instantiation method*), 51

`setFirstOut()` (*pyAgrum.Instantiation method*), 51

`setFirstVar()` (*pyAgrum.Instantiation method*), 51

`setForbiddenArcs()` (*pyAgrum.BN Learner method*), 186

`setInitialDAG()` (*pyAgrum.BN Learner method*), 186

`setLast()` (*pyAgrum.Instantiation method*), 51

`setLastIn()` (*pyAgrum.Instantiation method*), 51

`setLastNotVar()` (*pyAgrum.Instantiation method*), 52

`setLastOut()` (*pyAgrum.Instantiation method*), 52

`setLastVar()` (*pyAgrum.Instantiation method*), 52

`setMandatoryArcs()` (*pyAgrum.BN Learner method*), 186

`setMaxIndegree()` (*pyAgrum.BN Learner method*), 187

`setMaxIter()` (*pyAgrum.BN Learner method*), 187

`setMaxIter()` (*pyAgrum.CNLoopyPropagation method*), 215

`setMaxIter()` (*pyAgrum.CNMonteCarloSampling method*), 211

`setMaxIter()` (*pyAgrum.GibbsBNdistance method*), 84

`setMaxIter()` (*pyAgrum.GibbsSampling method*), 132

`setMaxIter()` (*pyAgrum.ImportanceSampling method*), 152

`setMaxIter()` (*pyAgrum.LoopyBeliefPropagation method*), 125

`setMaxIter()` (*pyAgrum.LoopyGibbsSampling method*), 160

`setMaxIter()` (*pyAgrum.LoopyImportanceSampling method*), 180

`setMaxIter()` (*pyAgrum.LoopyMonteCarloSampling method*), 167

`setMaxIter()` (*pyAgrum.LoopyWeightedSampling method*), 173

`setMaxIter()` (*pyAgrum.MonteCarloSampling method*), 139

`setMaxIter()` (*pyAgrum.WeightedSampling method*), 146

`setMaxMemory()` (*pyAgrum.LazyPropagation method*), 104

`setMaxMemory()` (*pyAgrum.ShaferShenoyInference method*), 112

`setMaxMemory()` (*pyAgrum.ShaferShenoyMNIInference method*), 229

`setMaxMemory()` (*pyAgrum.VariableElimination method*), 118

`setMaxTime()` (*pyAgrum.BN Learner method*), 187

`setMaxTime()` (*pyAgrum.CNLoopyPropagation method*), 215



`setMaxTime()` (*pyAgrum.CNMonteCarloSampling method*), 211  
`setMaxTime()` (*pyAgrum.GibbsBNdistance method*), 84  
`setMaxTime()` (*pyAgrum.GibbsSampling method*), 132  
`setMaxTime()` (*pyAgrum.ImportanceSampling method*), 153  
`setMaxTime()` (*pyAgrum.LoopyBeliefPropagation method*), 125  
`setMaxTime()` (*pyAgrum.LoopyGibbsSampling method*), 160  
`setMaxTime()` (*pyAgrum.LoopyImportanceSampling method*), 180  
`setMaxTime()` (*pyAgrum.LoopyMonteCarloSampling method*), 167  
`setMaxTime()` (*pyAgrum.LoopyWeightedSampling method*), 174  
`setMaxTime()` (*pyAgrum.MonteCarloSampling method*), 139  
`setMaxTime()` (*pyAgrum.WeightedSampling method*), 146  
`setMaxVal()` (*pyAgrum.RangeVariable method*), 41  
`setMinEpsilonRate()` (*pyAgrum.BN Learner method*), 187  
`setMinEpsilonRate()` (*pyAgrum.CNLoopyPropagation method*), 215  
`setMinEpsilonRate()` (*pyAgrum.CNMonteCarloSampling method*), 212  
`setMinEpsilonRate()` (*pyAgrum.GibbsBNdistance method*), 84  
`setMinEpsilonRate()` (*pyAgrum.GibbsSampling method*), 132  
`setMinEpsilonRate()` (*pyAgrum.ImportanceSampling method*), 153  
`setMinEpsilonRate()` (*pyAgrum.LoopyBeliefPropagation method*), 125  
`setMinEpsilonRate()` (*pyAgrum.LoopyGibbsSampling method*), 160  
`setMinEpsilonRate()` (*pyAgrum.LoopyImportanceSampling method*), 180  
`setMinEpsilonRate()` (*pyAgrum.LoopyMonteCarloSampling method*), 167  
`setMinEpsilonRate()` (*pyAgrum.LoopyWeightedSampling method*), 174  
`setMinEpsilonRate()` (*pyAgrum.MonteCarloSampling method*), 139  
`setMinEpsilonRate()` (*pyAgrum.WeightedSampling method*), 146  
`setMinVal()` (*pyAgrum.RangeVariable method*), 41  
`setMutable()` (*pyAgrum.Instantiation method*), 52  
`setName()` (*pyAgrum.DiscreteVariable method*), 26  
`setName()` (*pyAgrum.DiscretizedVariable method*), 33  
`setName()` (*pyAgrum.IntegerVariable method*), 37  
`setName()` (*pyAgrum.LabelizedVariable method*), 30  
`setName()` (*pyAgrum.NumericalDiscreteVariable method*), 44  
`setName()` (*pyAgrum.RangeVariable method*), 41  
`setNbrDrawnVar()` (*pyAgrum.GibbsBNdistance method*), 84  
`setNbrDrawnVar()` (*pyAgrum.GibbsSampling method*), 132  
`setNbrDrawnVar()` (*pyAgrum.LoopyGibbsSampling method*), 160  
`setNumberOfThreads()` (*in module pyAgrum*), 286  
`setNumberOfThreads()` (*pyAgrum.BN Learner method*), 187  
`setNumberOfThreads()` (*pyAgrum.LazyPropagation method*), 105  
`setNumberOfThreads()` (*pyAgrum.ShaferShenoyInference method*), 112  
`setNumberOfThreads()` (*pyAgrum.ShaferShenoyMNIInference method*), 229  
`setNumberOfThreads()` (*pyAgrum.VariableElimination method*), 119  
`setPeriodSize()` (*pyAgrum.BN Learner method*), 187  
`setPeriodSize()` (*pyAgrum.CNLoopyPropagation method*), 215  
`setPeriodSize()` (*pyAgrum.CNMonteCarloSampling method*), 212  
`setPeriodSize()` (*pyAgrum.GibbsBNdistance method*), 84  
`setPeriodSize()` (*pyAgrum.GibbsSampling method*), 133  
`setPeriodSize()` (*pyAgrum.ImportanceSampling method*), 153  
`setPeriodSize()` (*pyAgrum.LoopyBeliefPropagation method*), 125  
`setPeriodSize()` (*pyAgrum.LoopyGibbsSampling method*), 160  
`setPeriodSize()` (*pyAgrum.LoopyImportanceSampling method*), 181  
`setPeriodSize()` (*pyAgrum.LoopyMonteCarloSampling method*), 167  
`setPeriodSize()` (*pyAgrum.LoopyWeightedSampling method*), 174  
`setPeriodSize()` (*pyAgrum.MonteCarloSampling method*), 139  
`setPeriodSize()` (*pyAgrum.WeightedSampling method*), 146  
`setPossibleEdges()` (*pyAgrum.BN Learner method*), 187  
`setPossibleSkeleton()` (*pyAgrum.BN Learner method*), 187

- `setProperty()` (*pyAgrum.BayesNetFragment* method), 96
- `setRandomVarOrder()` (*pyAgrum.BNDatabaseGenerator* method), 79
- `setRecordWeight()` (*pyAgrum.BN Learner* method), 187
- `setRepetitiveInd()` (*pyAgrum.CN LoopyPropagation* method), 215
- `setRepetitiveInd()` (*pyAgrum.CN MonteCarloSampling* method), 212
- `setSliceOrder()` (*pyAgrum.BN Learner* method), 188
- `setTargets()` (*pyAgrum.GibbsSampling* method), 133
- `setTargets()` (*pyAgrum.ImportanceSampling* method), 153
- `setTargets()` (*pyAgrum.LazyPropagation* method), 105
- `setTargets()` (*pyAgrum.LoopyBeliefPropagation* method), 125
- `setTargets()` (*pyAgrum.LoopyGibbsSampling* method), 160
- `setTargets()` (*pyAgrum.LoopyImportanceSampling* method), 181
- `setTargets()` (*pyAgrum.LoopyMonteCarloSampling* method), 167
- `setTargets()` (*pyAgrum.LoopyWeightedSampling* method), 174
- `setTargets()` (*pyAgrum.MonteCarloSampling* method), 139
- `setTargets()` (*pyAgrum.ShaferShenoyInference* method), 112
- `setTargets()` (*pyAgrum.ShaferShenoyMNI nference* method), 229
- `setTargets()` (*pyAgrum.VariableElimination* method), 119
- `setTargets()` (*pyAgrum.WeightedSampling* method), 146
- `setTopologicalVarOrder()` (*pyAgrum.BNDatabaseGenerator* method), 79
- `setVals()` (*pyAgrum.Instantiation* method), 52
- `setVarOrder()` (*pyAgrum.BNDatabaseGenerator* method), 79
- `setVarOrderFromCSV()` (*pyAgrum.BNDatabaseGenerator* method), 79
- `setVerbosity()` (*pyAgrum.BN Learner* method), 188
- `setVerbosity()` (*pyAgrum.CN LoopyPropagation* method), 216
- `setVerbosity()` (*pyAgrum.CN MonteCarloSampling* method), 212
- `setVerbosity()` (*pyAgrum.GibbsBN distance* method), 84
- `setVerbosity()` (*pyAgrum.GibbsSampling* method), 133
- `setVerbosity()` (*pyAgrum.ImportanceSampling* method), 153
- `setVerbosity()` (*pyAgrum.LoopyBeliefPropagation* method), 125
- `setVerbosity()` (*pyAgrum.LoopyGibbsSampling* method), 160
- `setVerbosity()` (*pyAgrum.LoopyImportanceSampling* method), 181
- `setVerbosity()` (*pyAgrum.LoopyMonteCarloSampling* method), 167
- `setVerbosity()` (*pyAgrum.LoopyWeightedSampling* method), 174
- `setVerbosity()` (*pyAgrum.MonteCarloSampling* method), 140
- `setVerbosity()` (*pyAgrum.WeightedSampling* method), 146
- `setVirtualBPSize()` (*pyAgrum.LoopyGibbsSampling* method), 161
- `setVirtualBPSize()` (*pyAgrum.LoopyImportanceSampling* method), 181
- `setVirtualBPSize()` (*pyAgrum.LoopyMonteCarloSampling* method), 167
- `setVirtualBPSize()` (*pyAgrum.LoopyWeightedSampling* method), 174
- `ShaferShenoyInference` (class in *pyAgrum*), 105
- `ShaferShenoyLIMIDInference` (class in *pyAgrum*), 200
- `ShaferShenoyMNI nference` (class in *pyAgrum*), 223
- `shape` (*pyAgrum.Potential* property), 61
- `showBN()` (in module *pyAgrum.lib.notebook*), 262
- `showCausalImpact()` (in module *pyAgrum.causal.notebook*), 253
- `showCausalModel()` (in module *pyAgrum.causal.notebook*), 253
- `showCN()` (in module *pyAgrum.lib.notebook*), 264
- `showDot()` (in module *pyAgrum.lib.notebook*), 268
- `showGraph()` (in module *pyAgrum.lib.notebook*), 268
- `showInference()` (in module *pyAgrum.lib.notebook*), 265
- `showInfluenceDiagram()` (in module *pyAgrum.lib.notebook*), 263
- `showInformation()` (in module *pyAgrum.lib.explain*), 271
- `showJunctionTree()` (in module *pyAgrum.lib.notebook*), 266
- `showMNI nference` (in module *pyAgrum.lib.notebook*), 263
- `showPosterior()` (in module *pyAgrum.lib.notebook*), 267
- `showPotential()` (in module *pyAgrum.lib.notebook*), 267
- `showProba()` (in module *pyAgrum.lib.notebook*), 267
- `showROC_PR()` (*pyAgrum.skbn.BNClassifier* method), 258

- sideBySide() (in module *pyAgrum.lib.notebook*), 269  
 size() (*pyAgrum.BayesNet* method), 77  
 size() (*pyAgrum.BayesNetFragment* method), 96  
 size() (*pyAgrum.CliqueGraph* method), 19  
 size() (*pyAgrum.DAG* method), 11  
 size() (*pyAgrum.DiGraph* method), 7  
 size() (*pyAgrum.EssentialGraph* method), 87  
 size() (*pyAgrum.InfluenceDiagram* method), 199  
 size() (*pyAgrum.MarkovBlanket* method), 89  
 size() (*pyAgrum.MarkovNet* method), 222  
 size() (*pyAgrum.MixedGraph* method), 24  
 size() (*pyAgrum.UndiGraph* method), 14  
 sizeArcs() (*pyAgrum.BayesNet* method), 77  
 sizeArcs() (*pyAgrum.BayesNetFragment* method), 96  
 sizeArcs() (*pyAgrum.DAG* method), 11  
 sizeArcs() (*pyAgrum.DiGraph* method), 7  
 sizeArcs() (*pyAgrum.EssentialGraph* method), 87  
 sizeArcs() (*pyAgrum.InfluenceDiagram* method), 199  
 sizeArcs() (*pyAgrum.MarkovBlanket* method), 89  
 sizeArcs() (*pyAgrum.MixedGraph* method), 24  
 sizeEdges() (*pyAgrum.CliqueGraph* method), 19  
 sizeEdges() (*pyAgrum.EssentialGraph* method), 87  
 sizeEdges() (*pyAgrum.MarkovNet* method), 222  
 sizeEdges() (*pyAgrum.MixedGraph* method), 24  
 sizeEdges() (*pyAgrum.UndiGraph* method), 14  
 SizeError, 290  
 sizeNodes() (*pyAgrum.EssentialGraph* method), 88  
 sizeNodes() (*pyAgrum.MarkovBlanket* method), 89  
 skeleton() (*pyAgrum.EssentialGraph* method), 88  
 smallestFactorFromNode() (*pyAgrum.MarkovNet* method), 222  
 softEvidenceNodes() (*pyAgrum.GibbsSampling* method), 133  
 softEvidenceNodes() (*pyAgrum.ImportanceSampling* method), 153  
 softEvidenceNodes() (*pyAgrum.LazyPropagation* method), 105  
 softEvidenceNodes() (*pyAgrum.LoopyBeliefPropagation* method), 125  
 softEvidenceNodes() (*pyAgrum.LoopyGibbsSampling* method), 161  
 softEvidenceNodes() (*pyAgrum.LoopyImportanceSampling* method), 181  
 softEvidenceNodes() (*pyAgrum.LoopyMonteCarloSampling* method), 167  
 softEvidenceNodes() (*pyAgrum.LoopyWeightedSampling* method), 174  
 softEvidenceNodes() (*pyAgrum.MonteCarloSampling* method), 140  
 softEvidenceNodes() (*pyAgrum.ShaferShenoyInference* method), 112  
 softEvidenceNodes() (*pyAgrum.ShaferShenoyLIMIDInference* method), 203  
 softEvidenceNodes() (*pyAgrum.ShaferShenoyMNIInference* method), 230  
 softEvidenceNodes() (*pyAgrum.VariableElimination* method), 119  
 softEvidenceNodes() (*pyAgrum.WeightedSampling* method), 146  
 sq() (*pyAgrum.Potential* method), 61  
 src\_bn() (*pyAgrum.CredalNet* method), 209  
 startOfPeriod() (*pyAgrum.GibbsBNdistance* method), 84  
 state() (*pyAgrum.BN Learner* method), 188  
 stateApproximationScheme() (*pyAgrum.GibbsBNdistance* method), 84  
 stopApproximationScheme() (*pyAgrum.GibbsBNdistance* method), 85  
 stype() (*pyAgrum.DiscreteVariable* method), 26  
 stype() (*pyAgrum.DiscretizedVariable* method), 34  
 stype() (*pyAgrum.IntegerVariable* method), 37  
 stype() (*pyAgrum.LabelizedVariable* method), 30  
 stype() (*pyAgrum.NumericalDiscreteVariable* method), 44  
 stype() (*pyAgrum.RangeVariable* method), 41  
 sum() (*pyAgrum.Potential* method), 61  
 SyntaxError, 290
- ## T
- tail() (*pyAgrum.Arc* method), 4  
 targets() (*pyAgrum.GibbsSampling* method), 133  
 targets() (*pyAgrum.ImportanceSampling* method), 153  
 targets() (*pyAgrum.LazyPropagation* method), 105  
 targets() (*pyAgrum.LoopyBeliefPropagation* method), 126  
 targets() (*pyAgrum.LoopyGibbsSampling* method), 161  
 targets() (*pyAgrum.LoopyImportanceSampling* method), 181  
 targets() (*pyAgrum.LoopyMonteCarloSampling* method), 167  
 targets() (*pyAgrum.LoopyWeightedSampling* method), 174  
 targets() (*pyAgrum.MonteCarloSampling* method), 140  
 targets() (*pyAgrum.ShaferShenoyInference* method), 112  
 targets() (*pyAgrum.ShaferShenoyMNIInference* method), 230  
 targets() (*pyAgrum.VariableElimination* method), 119  
 targets() (*pyAgrum.WeightedSampling* method), 147  
 term (*pyAgrum.causal.ASTsum* property), 249  
 thisown (*pyAgrum.ArgumentError* property), 290  
 thisown (*pyAgrum.BayesNet* property), 77  
 thisown (*pyAgrum.CNLoopyPropagation* property), 216

- `thisown` (`pyAgrum.CPTError` property), 292
- `thisown` (`pyAgrum.DatabaseError` property), 292
- `thisown` (`pyAgrum.DefaultInLabel` property), 287
- `thisown` (`pyAgrum.DuplicateElement` property), 287
- `thisown` (`pyAgrum.DuplicateLabel` property), 287
- `thisown` (`pyAgrum.FatalError` property), 287
- `thisown` (`pyAgrum.FormatNotFound` property), 287
- `thisown` (`pyAgrum.GibbsSampling` property), 133
- `thisown` (`pyAgrum.GraphError` property), 288
- `thisown` (`pyAgrum.ImportanceSampling` property), 153
- `thisown` (`pyAgrum.InfluenceDiagram` property), 199
- `thisown` (`pyAgrum.InvalidArc` property), 288
- `thisown` (`pyAgrum.InvalidArgument` property), 288
- `thisown` (`pyAgrum.InvalidArgumentsNumber` property), 288
- `thisown` (`pyAgrum.InvalidDirectedCycle` property), 288
- `thisown` (`pyAgrum.InvalidEdge` property), 289
- `thisown` (`pyAgrum.InvalidNode` property), 289
- `thisown` (`pyAgrum.IOError` property), 288
- `thisown` (`pyAgrum.LazyPropagation` property), 105
- `thisown` (`pyAgrum.LoopyBeliefPropagation` property), 126
- `thisown` (`pyAgrum.LoopyGibbsSampling` property), 161
- `thisown` (`pyAgrum.LoopyImportanceSampling` property), 181
- `thisown` (`pyAgrum.LoopyMonteCarloSampling` property), 168
- `thisown` (`pyAgrum.LoopyWeightedSampling` property), 174
- `thisown` (`pyAgrum.MarkovNet` property), 222
- `thisown` (`pyAgrum.MonteCarloSampling` property), 140
- `thisown` (`pyAgrum.NoChild` property), 289
- `thisown` (`pyAgrum.NoNeighbour` property), 289
- `thisown` (`pyAgrum.NoParent` property), 289
- `thisown` (`pyAgrum.NotFound` property), 290
- `thisown` (`pyAgrum.NullElement` property), 290
- `thisown` (`pyAgrum.OperationNotAllowed` property), 290
- `thisown` (`pyAgrum.OutOfBounds` property), 290
- `thisown` (`pyAgrum.Potential` property), 61
- `thisown` (`pyAgrum.ShaferShenoyInference` property), 112
- `thisown` (`pyAgrum.ShaferShenoyMNIInference` property), 230
- `thisown` (`pyAgrum.SizeError` property), 290
- `thisown` (`pyAgrum.SyntaxError` property), 291
- `thisown` (`pyAgrum.UndefinedElement` property), 291
- `thisown` (`pyAgrum.UndefinedIteratorKey` property), 291
- `thisown` (`pyAgrum.UndefinedIteratorValue` property), 291
- `thisown` (`pyAgrum.UnknownLabelInDatabase` property), 292
- `thisown` (`pyAgrum.VariableElimination` property), 119
- `thisown` (`pyAgrum.WeightedSampling` property), 147
- `tick()` (`pyAgrum.DiscretizedVariable` method), 34
- `ticks()` (`pyAgrum.DiscretizedVariable` method), 34
- `to_pandas()` (`pyAgrum.BNDatabaseGenerator` method), 79
- `toarray()` (`pyAgrum.Potential` method), 61
- `toBN()` (`pyAgrum.BayesNetFragment` method), 96
- `toclipboard()` (`pyAgrum.Potential` method), 61
- `toCSV()` (`pyAgrum.BNDatabaseGenerator` method), 79
- `toDatabaseTable()` (`pyAgrum.BNDatabaseGenerator` method), 79
- `todict()` (`pyAgrum.Instantiation` method), 52
- `toDiscretizedVar()` (`pyAgrum.DiscreteVariable` method), 26
- `toDiscretizedVar()` (`pyAgrum.DiscretizedVariable` method), 34
- `toDiscretizedVar()` (`pyAgrum.IntegerVariable` method), 38
- `toDiscretizedVar()` (`pyAgrum.LabelizedVariable` method), 30
- `toDiscretizedVar()` (`pyAgrum.NumericalDiscreteVariable` method), 44
- `toDiscretizedVar()` (`pyAgrum.RangeVariable` method), 41
- `toDot()` (`pyAgrum.BayesNet` method), 77
- `toDot()` (`pyAgrum.BayesNetFragment` method), 96
- `toDot()` (`pyAgrum.causal.CausalModel` method), 238
- `toDot()` (`pyAgrum.CliqueGraph` method), 19
- `toDot()` (`pyAgrum.DAG` method), 11
- `toDot()` (`pyAgrum.DiGraph` method), 8
- `toDot()` (`pyAgrum.EssentialGraph` method), 88
- `toDot()` (`pyAgrum.InfluenceDiagram` method), 199
- `toDot()` (`pyAgrum.MarkovBlanket` method), 89
- `toDot()` (`pyAgrum.MarkovNet` method), 222
- `toDot()` (`pyAgrum.MixedGraph` method), 24
- `toDot()` (`pyAgrum.UndiGraph` method), 14
- `toDotAsFactorGraph()` (`pyAgrum.MarkovNet` method), 222
- `toDotWithNames()` (`pyAgrum.CliqueGraph` method), 19
- `toIntegerVar()` (`pyAgrum.DiscreteVariable` method), 26
- `toIntegerVar()` (`pyAgrum.DiscretizedVariable` method), 34
- `toIntegerVar()` (`pyAgrum.IntegerVariable` method), 38
- `toIntegerVar()` (`pyAgrum.LabelizedVariable` method), 30
- `toIntegerVar()` (`pyAgrum.NumericalDiscreteVariable` method), 45
- `toIntegerVar()` (`pyAgrum.RangeVariable` method), 41
- `toLabelizedVar()` (`pyAgrum.DiscreteVariable` method), 26



- [toLabelizedVar\(\)](#) ([pyAgrum.DiscretizedVariable](#) method), 34  
[toLabelizedVar\(\)](#) ([pyAgrum.IntegerVariable](#) method), 38  
[toLabelizedVar\(\)](#) ([pyAgrum.LabelizedVariable](#) method), 30  
[toLabelizedVar\(\)](#) ([pyAgrum.NumericalDiscreteVariable](#) method), 45  
[toLabelizedVar\(\)](#) ([pyAgrum.RangeVariable](#) method), 42  
[toLatex\(\)](#) ([pyAgrum.causal.ASTBinaryOp](#) method), 243  
[toLatex\(\)](#) ([pyAgrum.causal.ASTdiv](#) method), 247  
[toLatex\(\)](#) ([pyAgrum.causal.ASTjointProba](#) method), 250  
[toLatex\(\)](#) ([pyAgrum.causal.ASTminus](#) method), 246  
[toLatex\(\)](#) ([pyAgrum.causal.ASTmult](#) method), 248  
[toLatex\(\)](#) ([pyAgrum.causal.ASTplus](#) method), 244  
[toLatex\(\)](#) ([pyAgrum.causal.ASTposteriorProba](#) method), 252  
[toLatex\(\)](#) ([pyAgrum.causal.ASTsum](#) method), 249  
[toLatex\(\)](#) ([pyAgrum.causal.ASTtree](#) method), 242  
[toLatex\(\)](#) ([pyAgrum.causal.CausalFormula](#) method), 239  
[tolatex\(\)](#) ([pyAgrum.Potential](#) method), 61  
[tolist\(\)](#) ([pyAgrum.Potential](#) method), 62  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.DiscreteVariable](#) method), 27  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.DiscretizedVariable](#) method), 34  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.IntegerVariable](#) method), 38  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.LabelizedVariable](#) method), 30  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.NumericalDiscreteVariable](#) method), 45  
[toNumericalDiscreteVar\(\)](#) ([pyAgrum.RangeVariable](#) method), 42  
[topandas\(\)](#) ([pyAgrum.Potential](#) method), 62  
[topologicalOrder\(\)](#) ([pyAgrum.BayesNet](#) method), 77  
[topologicalOrder\(\)](#) ([pyAgrum.BayesNetFragment](#) method), 96  
[topologicalOrder\(\)](#) ([pyAgrum.DAG](#) method), 11  
[topologicalOrder\(\)](#) ([pyAgrum.DiGraph](#) method), 8  
[topologicalOrder\(\)](#) ([pyAgrum.InfluenceDiagram](#) method), 199  
[topologicalOrder\(\)](#) ([pyAgrum.MixedGraph](#) method), 24  
[toRangeVar\(\)](#) ([pyAgrum.DiscreteVariable](#) method), 27  
[toRangeVar\(\)](#) ([pyAgrum.DiscretizedVariable](#) method), 34  
[toRangeVar\(\)](#) ([pyAgrum.IntegerVariable](#) method), 38  
[toRangeVar\(\)](#) ([pyAgrum.LabelizedVariable](#) method), 31  
[toRangeVar\(\)](#) ([pyAgrum.NumericalDiscreteVariable](#) method), 45  
[toRangeVar\(\)](#) ([pyAgrum.RangeVariable](#) method), 42  
[toStringWithDescription\(\)](#) ([pyAgrum.DiscreteVariable](#) method), 27  
[toStringWithDescription\(\)](#) ([pyAgrum.DiscretizedVariable](#) method), 34  
[toStringWithDescription\(\)](#) ([pyAgrum.IntegerVariable](#) method), 38  
[toStringWithDescription\(\)](#) ([pyAgrum.LabelizedVariable](#) method), 31  
[toStringWithDescription\(\)](#) ([pyAgrum.NumericalDiscreteVariable](#) method), 45  
[toStringWithDescription\(\)](#) ([pyAgrum.RangeVariable](#) method), 42  
[translate\(\)](#) ([pyAgrum.Potential](#) method), 62  
[type](#) ([pyAgrum.causal.ASTBinaryOp](#) property), 243  
[type](#) ([pyAgrum.causal.ASTdiv](#) property), 247  
[type](#) ([pyAgrum.causal.ASTjointProba](#) property), 250  
[type](#) ([pyAgrum.causal.ASTminus](#) property), 246  
[type](#) ([pyAgrum.causal.ASTmult](#) property), 248  
[type](#) ([pyAgrum.causal.ASTplus](#) property), 245  
[type](#) ([pyAgrum.causal.ASTposteriorProba](#) property), 252  
[type](#) ([pyAgrum.causal.ASTsum](#) property), 249  
[type](#) ([pyAgrum.causal.ASTtree](#) property), 242  
[types\(\)](#) ([pyAgrum.PRMexplorer](#) method), 235
- ## U
- [UndefinedElement](#), 291  
[UndefinedIteratorKey](#), 291  
[UndefinedIteratorValue](#), 291  
[UndiGraph](#) (class in [pyAgrum](#)), 12  
[UnidentifiableException](#) (class in [pyAgrum.causal](#)), 252  
[uninstallCPT\(\)](#) ([pyAgrum.BayesNetFragment](#) method), 96  
[uninstallNode\(\)](#) ([pyAgrum.BayesNetFragment](#) method), 96  
[UnknownLabelInDatabase](#), 291  
[unsetEnd\(\)](#) ([pyAgrum.Instantiation](#) method), 52  
[unsetOverflow\(\)](#) ([pyAgrum.Instantiation](#) method), 52  
[updateApproximationScheme\(\)](#) ([pyAgrum.GibbsBNdistance](#) method), 85  
[updateEvidence\(\)](#) ([pyAgrum.GibbsSampling](#) method), 133  
[updateEvidence\(\)](#) ([pyAgrum.ImportanceSampling](#) method), 153  
[updateEvidence\(\)](#) ([pyAgrum.LazyPropagation](#) method), 105  
[updateEvidence\(\)](#) ([pyAgrum.LoopyBeliefPropagation](#) method), 126  
[updateEvidence\(\)](#) ([pyAgrum.LoopyGibbsSampling](#) method), 161

- updateEvidence() (pyAgrum.LoopyImportanceSampling method), 181
- updateEvidence() (pyAgrum.LoopyMonteCarloSampling method), 168
- updateEvidence() (pyAgrum.LoopyWeightedSampling method), 175
- updateEvidence() (pyAgrum.MonteCarloSampling method), 140
- updateEvidence() (pyAgrum.ShaferShenoyInference method), 112
- updateEvidence() (pyAgrum.ShaferShenoyLIMIDInference method), 203
- updateEvidence() (pyAgrum.ShaferShenoyMNInference method), 230
- updateEvidence() (pyAgrum.VariableElimination method), 119
- updateEvidence() (pyAgrum.WeightedSampling method), 147
- use3off2() (pyAgrum.BN Learner method), 188
- useAprioriBDeu() (pyAgrum.BN Learner method), 188
- useAprioriDirichlet() (pyAgrum.BN Learner method), 188
- useAprioriSmoothing() (pyAgrum.BN Learner method), 188
- useBDeuPrior() (pyAgrum.BN Learner method), 188
- useDirichletPrior() (pyAgrum.BN Learner method), 188
- useEM() (pyAgrum.BN Learner method), 188
- useGreedyHillClimbing() (pyAgrum.BN Learner method), 188
- useK2() (pyAgrum.BN Learner method), 188
- useLocalSearchWithTabuList() (pyAgrum.BN Learner method), 189
- useMDLCorrection() (pyAgrum.BN Learner method), 189
- useMIIC() (pyAgrum.BN Learner method), 189
- useNMLCorrection() (pyAgrum.BN Learner method), 189
- useNoApriori() (pyAgrum.BN Learner method), 189
- useNoCorrection() (pyAgrum.BN Learner method), 189
- useNoPrior() (pyAgrum.BN Learner method), 189
- useScoreAIC() (pyAgrum.BN Learner method), 189
- useScoreBD() (pyAgrum.BN Learner method), 189
- useScoreBDeu() (pyAgrum.BN Learner method), 189
- useScoreBIC() (pyAgrum.BN Learner method), 189
- useScoreK2() (pyAgrum.BN Learner method), 189
- useScoreLog2Likelihood() (pyAgrum.BN Learner method), 190
- useSmoothingPrior() (pyAgrum.BN Learner method), 190
- utility() (pyAgrum.InfluenceDiagram method), 199
- utilityNodeSize() (pyAgrum.InfluenceDiagram method), 200
- ## V
- val() (pyAgrum.Instantiation method), 53
- var\_dims (pyAgrum.Potential property), 62
- var\_names (pyAgrum.Potential property), 62
- variable() (pyAgrum.BayesNet method), 77
- variable() (pyAgrum.BayesNetFragment method), 97
- variable() (pyAgrum.InfluenceDiagram method), 200
- variable() (pyAgrum.Instantiation method), 53
- variable() (pyAgrum.MarkovNet method), 222
- variable() (pyAgrum.Potential method), 62
- VariableElimination (class in pyAgrum), 113
- variableFromName() (pyAgrum.BayesNet method), 78
- variableFromName() (pyAgrum.BayesNetFragment method), 97
- variableFromName() (pyAgrum.InfluenceDiagram method), 200
- variableFromName() (pyAgrum.MarkovNet method), 222
- variableNodeMap() (pyAgrum.BayesNet method), 78
- variableNodeMap() (pyAgrum.BayesNetFragment method), 97
- variableNodeMap() (pyAgrum.InfluenceDiagram method), 200
- variableNodeMap() (pyAgrum.MarkovNet method), 222
- variablesSequence() (pyAgrum.Instantiation method), 53
- variablesSequence() (pyAgrum.Potential method), 62
- varNames (pyAgrum.causal.ASTjointProba property), 251
- varOrder() (pyAgrum.BNDatabaseGenerator method), 79
- varOrderNames() (pyAgrum.BNDatabaseGenerator method), 79
- vars (pyAgrum.causal.ASTposteriorProba property), 252
- varType() (pyAgrum.DiscreteVariable method), 27
- varType() (pyAgrum.DiscretizedVariable method), 35
- varType() (pyAgrum.IntegerVariable method), 38
- varType() (pyAgrum.LabelizedVariable method), 31
- varType() (pyAgrum.NumericalDiscreteVariable method), 45
- varType() (pyAgrum.RangeVariable method), 42
- verbosity() (pyAgrum.BN Learner method), 190
- verbosity() (pyAgrum.CNLoopyPropagation method), 216
- verbosity() (pyAgrum.CNMonteCarloSampling method), 212
- verbosity() (pyAgrum.GibbsBNdistance method), 85
- verbosity() (pyAgrum.GibbsSampling method), 133
- verbosity() (pyAgrum.ImportanceSampling method), 154

`verbosity()` (*pyAgrum.LoopyBeliefPropagation method*), 126  
`verbosity()` (*pyAgrum.LoopyGibbsSampling method*), 161  
`verbosity()` (*pyAgrum.LoopyImportanceSampling method*), 181  
`verbosity()` (*pyAgrum.LoopyMonteCarloSampling method*), 168  
`verbosity()` (*pyAgrum.LoopyWeightedSampling method*), 175  
`verbosity()` (*pyAgrum.MonteCarloSampling method*), 140  
`verbosity()` (*pyAgrum.WeightedSampling method*), 147  
`VI()` (*pyAgrum.LazyPropagation method*), 98  
`VI()` (*pyAgrum.ShaferShenoyInference method*), 106  
`VI()` (*pyAgrum.ShaferShenoyMNIInference method*), 223

## W

`WeightedSampling` (*class in pyAgrum*), 140  
`what()` (*pyAgrum.ArgumentError method*), 290  
`what()` (*pyAgrum.CPTErrror method*), 292  
`what()` (*pyAgrum.DatabaseError method*), 292  
`what()` (*pyAgrum.DuplicateElement method*), 287  
`what()` (*pyAgrum.DuplicateLabel method*), 287  
`what()` (*pyAgrum.FatalError method*), 287  
`what()` (*pyAgrum.FormatNotFound method*), 287  
`what()` (*pyAgrum.GraphError method*), 288  
`what()` (*pyAgrum.GumException method*), 287  
`what()` (*pyAgrum.InvalidArc method*), 288  
`what()` (*pyAgrum.InvalidArgument method*), 288  
`what()` (*pyAgrum.InvalidArgumentsNumber method*), 288  
`what()` (*pyAgrum.InvalidDirectedCycle method*), 289  
`what()` (*pyAgrum.InvalidEdge method*), 289  
`what()` (*pyAgrum.InvalidNode method*), 289  
`what()` (*pyAgrum.IOError method*), 288  
`what()` (*pyAgrum.NoChild method*), 289  
`what()` (*pyAgrum.NoNeighbour method*), 289  
`what()` (*pyAgrum.NoParent method*), 289  
`what()` (*pyAgrum.NotFound method*), 290  
`what()` (*pyAgrum.NullElement method*), 290  
`what()` (*pyAgrum.OperationNotAllowed method*), 290  
`what()` (*pyAgrum.OutOfBounds method*), 290  
`what()` (*pyAgrum.SizeError method*), 290  
`what()` (*pyAgrum.SyntaxError method*), 291  
`what()` (*pyAgrum.UndefinedElement method*), 291  
`what()` (*pyAgrum.UndefinedIteratorKey method*), 291  
`what()` (*pyAgrum.UndefinedIteratorValue method*), 291  
`what()` (*pyAgrum.UnknownLabelInDatabase method*), 292  
`whenArcAdded()` (*pyAgrum.BayesNetFragment method*), 97  
`whenArcDeleted()` (*pyAgrum.BayesNetFragment method*), 97  
`whenNodeAdded()` (*pyAgrum.BayesNetFragment method*), 97  
`whenNodeDeleted()` (*pyAgrum.BayesNetFragment method*), 97  
`with_traceback()` (*pyAgrum.GumException method*), 287

## X

`XYfromCSV()` (*pyAgrum.skbn.BNClassifier method*), 256