

# **pyAgrum Documentation**

***Release 0.22.7***

**Pierre-Henri Wuillemin (Sphinx)**

**February 02, 2022**



# FUNDAMENTAL COMPONENTS

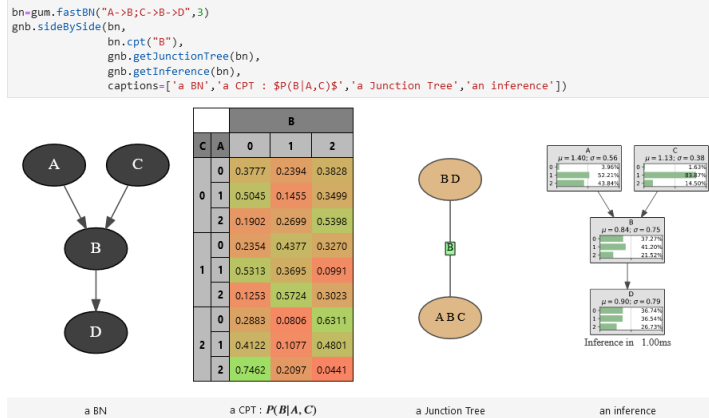
<b>1</b>	<b>Graphs manipulation</b>	<b>3</b>
1.1	Edges and Arcs . . . . .	3
1.2	Directed Graphs . . . . .	4
1.3	Undirected Graphs . . . . .	11
1.4	Mixed Graph . . . . .	18
<b>2</b>	<b>Random Variables</b>	<b>25</b>
2.1	Common API for Random Discrete Variables . . . . .	25
2.2	Concrete classes for Random Discrete Variables . . . . .	27
<b>3</b>	<b>Potential and Instantiation</b>	<b>41</b>
3.1	Instantiation . . . . .	42
3.2	Potential . . . . .	48
<b>4</b>	<b>Bayesian network</b>	<b>57</b>
4.1	Model . . . . .	58
4.2	Tools for Bayesian networks . . . . .	72
4.3	Inference . . . . .	90
4.4	Exact Inference . . . . .	90
4.5	Approximated Inference . . . . .	110
4.6	Learning . . . . .	173
<b>5</b>	<b>Influence Diagram</b>	<b>183</b>
5.1	Model . . . . .	184
5.2	Inference . . . . .	192
<b>6</b>	<b>Credal Network</b>	<b>197</b>
6.1	Model . . . . .	197
6.2	Inference . . . . .	202
<b>7</b>	<b>Markov Network</b>	<b>211</b>
7.1	Model . . . . .	212
7.2	Inference . . . . .	217
<b>8</b>	<b>Probabilistic Relational Models</b>	<b>227</b>
<b>9</b>	<b>pyAgrum.causal documentation</b>	<b>233</b>
9.1	Causal Model . . . . .	234
9.2	Causal Formula . . . . .	236
9.3	Causal Inference . . . . .	237
9.4	Abstract Syntax Tree for Do-Calculus . . . . .	238
9.5	Exceptions . . . . .	246
9.6	Notebook's tools for causality . . . . .	247
<b>10</b>	<b>pyAgrum.skbn documentation</b>	<b>249</b>

10.1	Classifier using Bayesian networks . . . . .	250
10.2	Discretizer for Bayesian networks . . . . .	253
<b>11</b>	<b>pyAgrum.lib.notebook</b>	<b>257</b>
11.1	Visualization of graphical models . . . . .	257
11.2	Visualization of Potentials . . . . .	262
11.3	Visualization of graphs . . . . .	263
11.4	Visualization of approximation algorithm . . . . .	264
11.5	Helpers . . . . .	264
<b>12</b>	<b>pyAgrum.lib.image</b>	<b>265</b>
12.1	Visualization of models and inference . . . . .	265
<b>13</b>	<b>pyAgrum.lib.explain</b>	<b>267</b>
13.1	Dealing with independence . . . . .	267
13.2	Dealing with mutual information and entropy . . . . .	267
13.3	Dealing with ShapValues . . . . .	268
<b>14</b>	<b>pyAgrum.lib.dynamicBN</b>	<b>271</b>
<b>15</b>	<b>other pyAgrum.lib modules</b>	<b>273</b>
15.1	bn2roc . . . . .	273
15.2	bn2scores . . . . .	274
15.3	bn_vs_bn . . . . .	274
<b>16</b>	<b>Functions from pyAgrum</b>	<b>277</b>
16.1	Useful functions in pyAgrum . . . . .	277
16.2	Quick specification of (randomly parameterized) graphical models . . . . .	277
16.3	Input/Output for Bayesian networks . . . . .	279
16.4	Input/Output for Markov networks . . . . .	280
16.5	Input for influence diagram . . . . .	281
<b>17</b>	<b>Other functions from aGrUM</b>	<b>283</b>
17.1	Listeners . . . . .	283
17.2	Random functions . . . . .	284
17.3	OMP functions . . . . .	285
<b>18</b>	<b>Exceptions from aGrUM</b>	<b>287</b>
<b>19</b>	<b>Configuration for pyAgrum</b>	<b>293</b>
<b>20</b>	<b>Indices and tables</b>	<b>295</b>
	<b>Python Module Index</b>	<b>297</b>
	<b>Index</b>	<b>299</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>)



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** (<https://pyagrum.readthedocs.io>).
- **tutorials as jupyter notebooks** (<http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/Tutorial.ipynb.html>).
- a **gitlab repository** (<https://gitlab.com/agrumery/aGrUM>).
- and a **website** (<http://agrum.org>).



## 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 58)), the objects are mapped to nodeIds.

### 1.1 Edges and Arcs

#### 1.1.1 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

## 1.1.2 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** (yAgrum.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

## 1.2 Directed Graphs

### 1.2.1 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 289) – If this graph contains cycles

**Parameters** **clear** (*bool*) –

## 1.2.2 Directed Acyclic Graph

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

DAG represents a Directed Acyclic Graph.

**DAG()** -> **DAG** default constructor

**DAG(src)** -> **DAG**

**Parameters:**

- **src** (*DAG*) – the DAG 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.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

**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 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

**moralGraph()**

**Return type** [UndiGraph](#) (page 11)

**moralizedAncestralGraph(*nodes*)**

**Parameters** **nodes** (*List[int]*) –

**Return type** [UndiGraph](#) (page 11)

**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 289) – If this graph contains cycles

**Parameters** **clear** (bool) –

## 1.3 Undirected Graphs

### 1.3.1 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** *pyAgrim.UndiGraph* (page 11)

**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

### 1.3.2 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 290) –
- **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 290) – 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 290) – 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 290) – 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 290) – 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 11)

**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 290) – 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 290) – 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 58)) –
- **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

## 1.4 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**(\**args*)

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**(\*args)

**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**(\*args)

**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 11)

**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** [pyAgrum.InvalidDirectedCycle](#) (page 289) – If this graph contains cycles

**Parameters** **clear** (bool) –



## RANDOM VARIABLES

aGrUM/pyAgrum is currently dedicated for discrete probability distributions.

There are 4 types of discrete random variables in aGrUM/pyAgrum: `LabelizedVariable`, `DiscretizedVariable`, `IntegerVariable` and `RangeVariable`. The 4 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.

### 2.1 Common API for Random Discrete Variables

**class** `pyAgrum.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()**

**Return type** str

**toDiscretizedVar()**

**Returns** the discretized variable

**Return type** [pyAgrum.DiscretizedVariable](#) (page 30)

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

**toIntegerVar()**

**Return type** [IntegerVariable](#) (page 33)

**toLabelizedVar()**

**Returns** the labelized variable

**Return type** [pyAgrum.LabelizedVariable](#) (page 27)

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

**toRangeVar()**

**Returns** the range variable

**Return type** *pyAgrum.RangeVariable* (page 36)

**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: RangeVariable

**Return type** int

## 2.2 Concrete classes for Random Discrete Variables

### 2.2.1 LabeledVariable

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

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

**LabeledVariable(aName, aDesc="", nbrLabel=2) -> LabeledVariable**

**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)

**LabeledVariable(aLDRV) -> LabeledVariable**

**Parameters:**

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

#### Examples

```
>>> import pyAgrum as gum
>>> # creating a variable with 3 labels : '0', '1' and '2'
>>> va=gum.LabeledVariable('a','a labeled variable',3)
>>> print(va)
a:Labelized(<0,1,2>)
>>> va.addLabel('foo')
("pyAgrum.LabeledVariable"@0x7fc4c840dd90) a:Labelized(<0,1,2,foo>)
>>> va.changeLabel(1,'bar')
>>> print(va)
a:Labelized(<0,bar,2,foo>)
>>> vb=gum.LabeledVariable('b','b',0).addLabel('A').addLabel('B').addLabel('C')
>>> print(vb)
b:Labelized(<A,B,C>)
>>> vb.labels()
('A', 'B', 'C')
>>> vb.isLabel('E')
```

(continues on next page)

(continued from previous page)

```
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 < 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()**

**Return type** str

**toDiscretizedVar()**

**Returns** the discretized variable

**Return type** *pyAgrum.DiscretizedVariable* (page 30)

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

**toIntegerVar()**

**Return type** *IntegerVariable* (page 33)

**toLabelizedVar()**

**Returns** the labeled variable

**Return type** *pyAgrum.LabelizedVariable* (page 27)

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

**toRangeVar()**

**Returns** the range variable

**Return type** *pyAgrum.RangeVariable* (page 36)

**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: RangeVariable

**Return type** int

## 2.2.2 DiscretizedVariable

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

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

**DiscretizedVariable(aName, aDesc="") -> DiscretizedVariable`**

**Parameters:**

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

**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 30)

**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

**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

**setName**(*theValue*)

sets the name of the variable.

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

**Return type** None

**stype**()

**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 290) – 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 30)

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

**toIntegerVar()**

**Return type** *IntegerVariable* (page 33)

**toLabelizedVar()**

**Returns** the labeled variable

**Return type** *pyAgrum.LabelizedVariable* (page 27)

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

**toRangeVar()**

**Returns** the range variable

**Return type** *pyAgrum.RangeVariable* (page 36)

**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: RangeVariable

**Return type** int

## 2.2.3 IntegerVariable

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

**addValue**(*value*)

**Parameters** *value* (int) –

**Return type** *IntegerVariable* (page 33)

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

**Parameters**

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

**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) –

**Return type** None

**eraseValues()**

**Return type** None

**index(*label*)**

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

**Returns** the indice of the label

**Return type** int

**integerDomain()**

**Return type** List[int]

**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()**

**Return type** str

**toDiscretizedVar()**

**Returns** the discretized variable

**Return type** *pyAgrum.DiscretizedVariable* (page 30)

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

**toIntegerVar()**

**Return type** *IntegerVariable* (page 33)

**toLabelizedVar()**

**Returns** the labelized variable

**Return type** *pyAgrum.LabelizedVariable* (page 27)

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

**toRangeVar()**

**Returns** the range variable

**Return type** *pyAgrum.RangeVariable* (page 36)

**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: RangeVariable

**Return type** int

## 2.2.4 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()**

**Return type** str

**toDiscretizedVar()**

**Returns** the discretized variable

**Return type** *pyAgrum.DiscretizedVariable* (page 30)

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

**toIntegerVar()**

**Return type** `IntegerVariable` (page 33)

**toLabelizedVar()**

**Returns** the labeled variable

**Return type** `pyAgrum.LabelizedVariable` (page 27)

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

**toRangeVar()**

**Returns** the range variable

**Return type** `pyAgrum.RangeVariable` (page 36)

**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: `RangeVariable`

**Return type** `int`



## POTENTIAL AND INSTANTIATION

*pyAgrum.Potential* (page 48) 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]);
>>> print(p)
<A:1|B:1> :: 1 /<A:2|B:1> :: 2 /<A:3|B:1> :: 3 /<A:1|B:2> :: 4 /<A:2|B:2> :: 5 /
↪<A:3|B:2> :: 6
```

- If a *pyAgrum.Potential* (page 48) 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:1|B:1> :: 0.166667 /<A:2|B:1> :: 0.333333 /<A:3|B:1> :: 0.5 /<A:1|B:2> :: 0.266667 ↪
↪<A:2|B:2> :: 0.333333 /<A:3|B:2> :: 0.4
```

- For addressing and looping in a *pyAgrum.Potential* (page 48) 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 48) 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:1|C:1|B:1> :: 2 /<A:2|C:1|B:1> :: 3 /<A:3|C:1|B:1> :: 4 /<A:1|C:2|B:1> :: 2 /
↪<A:2|C:2|B:1> :: 3 /<A:3|C:2|B:1> :: 4 /<A:1|C:3|B:1> :: 2 /<A:2|C:3|B:1> :: 3 /
↪<A:3|C:3|B:1> :: 4 /<A:1|C:4|B:1> :: 2 /<A:2|C:4|B:1> :: 3 /<A:3|C:4|B:1> :: 4 /
↪<A:1|C:5|B:1> :: 2 /<A:2|C:5|B:1> :: 3 /<A:3|C:5|B:1> :: 4 /<A:1|C:1|B:2> :: 5 /
↪<A:2|C:1|B:2> :: 6 /<A:3|C:1|B:2> :: 7 /<A:1|C:2|B:2> :: 5 /<A:2|C:2|B:2> :: 6 /
↪<A:3|C:2|B:2> :: 7 /<A:1|C:3|B:2> :: 5 /<A:2|C:3|B:2> :: 6 /<A:3|C:3|B:2> :: 7 /
↪<A:1|C:4|B:2> :: 5 /<A:2|C:4|B:2> :: 6 /<A:3|C:4|B:2> :: 7 /<A:1|C:5|B:2> :: 5 /
↪<A:2|C:5|B:2> :: 6 /<A:3|C:5|B:2> :: 7
```

(continued from previous page)

```
>>> print((p*q).margSumOut(["B","C"])) # marginalize p*q over B and C(using sum)
<A:1> :: 25 /<A:2> :: 35 /<A:3> :: 45
```

## 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 42)

**Raises**

- **NotFound** (page 290) – 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 42)) – 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 42)) – 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 290) – 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 290) – 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.

<b>Warning:</b> 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 42)) – 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 42)) – 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 290) – 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 290) – 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 42)) – 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 42)) – 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 42)) – 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 42)) – 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 42)) – 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 42)) – An Instantiation in which the new values are searched

**Returns** a reference to the instantiation

**Return type** [pyAgrum.Instantiation](#) (page 42)

**todict(withLabels=False)**

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** ([pyAgrum.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 290) – 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** [pyAgrum.DiscreteVariable](#) (page 25)

**Raises** **NotFound** (page 290) – If the element cannot be found.

**variablesSequence()**

**Returns** a list containing the sequence of variables

**Return type** list

## 3.2 Potential

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

Class representing a potential.

**Potential()** -> **Potential** default constructor

**Potential(src)** -> **Potential**

**Parameters:**

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

**KL(p)**

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

**Parameters** **p** (*pyAgrum.Potential* (page 48)) – the potential from which we want to calculate the divergence.

**Returns** The value of the divergence

**Return type** float

**Raises**

- *pyAgrum.InvalidArgument* (page 288) – If p is not compatible with the potential (dimension, variables)
- *pyAgrum.FatalError* (page 288) – 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 48)

**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 48)

**argmax()**

**Return type** List[Dict[str, int]]

**argmin()**

**Return type** List[Dict[str, int]]

**contains(v)**

**Parameters** **v** (*pyAgrum.Potential* (page 48)) – 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.

**Returns** the new Potential

**Return type** *pyAgrum.Potential* (page 48)

**fillWith(\*args)**

Automatically fills the potential with v.

**Parameters** **v** (*number or list or pyAgrum.Potential the number of parameters of the Potential*) – a value or a list/pyAgrum.Potential containing the values to fill the Potential with.

**Warning:** if v is a list, the size of the list must be the if v is a pyAgrum.Potential. It must to contain variables with exactly the same names and labels but not necessarily the same variables.

**Returns** a reference to the modified potential

**Return type** *pyAgrum.Potential* (page 48)

**Raises** *pyAgrum.SizeError* (page 290) – If v size's does not matches the domain size.

**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 48)

### 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 42)) – an Instantiation

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

**Return type** *float*

### `inverse()`

**Return type** *Potential* (page 48)

### `isNonZeroMap()`

**Returns** a boolean-like potential using the predicate `isNonZero`

**Return type** *pyAgrum.Potential* (page 48)

### `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 48)

**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 48)

**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 48)

**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 48)

**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 48)

**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 48)

**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 48)

**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 48)

**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 48)

**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 290) – 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 290) – If all value == 0.0

**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 48)

**new\_abs()**

**Return type** *Potential* (page 48)

**new\_log2()**

**Return type** *Potential* (page 48)

**new\_sq()**

**Return type** *Potential* (page 48)

**noising(*alpha*)**

**Parameters** *alpha* (float) –

**Return type** *Potential* (page 48)

**normalize()**

Normalize the Potential (do nothing if sum is 0)

**Returns** a reference to the normalized Potential

**Return type** *pyAgrum.Potential* (page 48)

**normalizeAsCPT(*varId=0*)**

Normalize the Potential as a CPT

**Returns** a reference to the normalized Potential

**Return type** *pyAgrum.Potential* (page 48)

**Raises** *pyAgrum.FatalError* (page 288) – 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 290) – 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 48)

Raises [\*pyAgrum.InvalidArgument\*](#) (page 288) – If the var is not in the potential

**random()**

**Return type** [\*Potential\*](#) (page 48)

**randomCPT()**

**Return type** [\*Potential\*](#) (page 48)

**randomDistribution()**

**Return type** [\*Potential\*](#) (page 48)

**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 48)

<b>Warning:</b> <code>IndexError</code> 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 48)

**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 42)) – The Instantiation to be changed
- **value** (*float*) – The new value of the Instantiation

**Return type** None

**sq()**

Square all the values in the Potential

**Return type** [\*Potential\*](#) (page 48)

**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

**property var\_names**

**Returns** a list containing the name of each variables in the potential

**Return type** list

<b>Warning:</b> listed 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 *i*th index

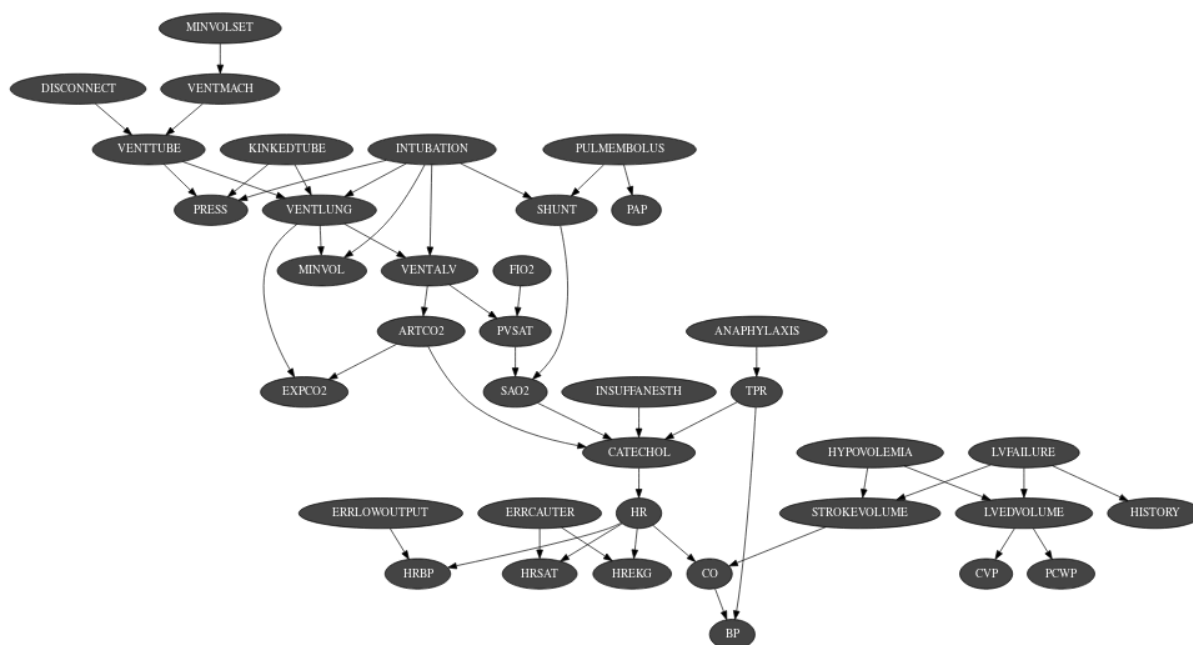
**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – If *i* does not reference a variable in this multidimensional matrix.

**variablesSequence()**

**Returns** a list containing the sequence of variables

**Return type** list

## BAYESIAN NETWORK



The Bayesian network is the main graphical model of pyAgrum. 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 pyAgrum, 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

- [Tutorial on Bayesian network](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Tutorial.ipynb.html) (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Tutorial.ipynb.html)

### Reference

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

**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** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)

**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

**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

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

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

**Parameters**

- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)
- **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**

- **newPot** ([pyAgrum.Potential](#) (page 48)) – the new potential
- **NodeId** (*int*) – the id of the node
- **name** (*str*) – the name of the variable

**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**

- **id** (*int*) – the id of the node
- **name** (*str*) – the name of the variable
- **old\_label** (*str*) – the new label
- **new\_label** (*str*) – the new label

**Raises** [pyAgrum.NotFound](#) (page 290) – 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**

- **new\_name** (*str*) – the new name of the variable
- **NodeId** (*int*) – the id of the node
- **name** (*str*) – the name of the variable

**Raises**

- [pyAgrum.DuplicateLabel](#) (page 287) – If new\_name is already used in this BayesNet.
- [pyAgrum.NotFound](#) (page 290) – If no variable matches id.

**Return type** None

**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 42)

**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.BayesNet.
- **name** (*str*) – A variable's name in the pyAgrum.BayesNet.

**Returns** The variable's CPT.

**Return type** *pyAgrum.Potential* (page 48)

**Raises** *pyAgrum.NotFound* (page 290) – 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 58)

**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**

- **id** (*int*) – The variable's id to remove.

- **name** (*str*) – The variable’s name to remove.
- **var** ([pyAgrum.DiscreteVariable](#) (page 25)) – A reference on the variable to remove.

**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](#) (page 3)) – The arc to be removed.
- **head** – a variable’s id (int)
- **tail** – a variable’s id (int)
- **head** – a variable’s name (str)
- **tail** – a variable’s name (str)

**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.

---

**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 58)

### **generateCPT**(\*args)

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

### Parameters

- **node** (*int*) – The variable's id.
- **name** (*str*) – The variable's 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 290) – 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 288) – 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 288) – 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 288) – 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 288) – 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 Discretized-Variable 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 288) – 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 288) – 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*) – The ids of the targets
- **list** (*list*) – The list of available variables

**Returns** The minimal set of variables

**Return type** Set

**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 11)

**Parameters** **clear** (bool) –

**moralizedAncestralGraph(nodes)**

**Parameters** **nodes** (object) –

**Return type** [UndiGraph](#) (page 11)

**names()**

**Returns** The names of the graph variables

**Return type** list

**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

**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 Discretized-Variable 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 289) – 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)

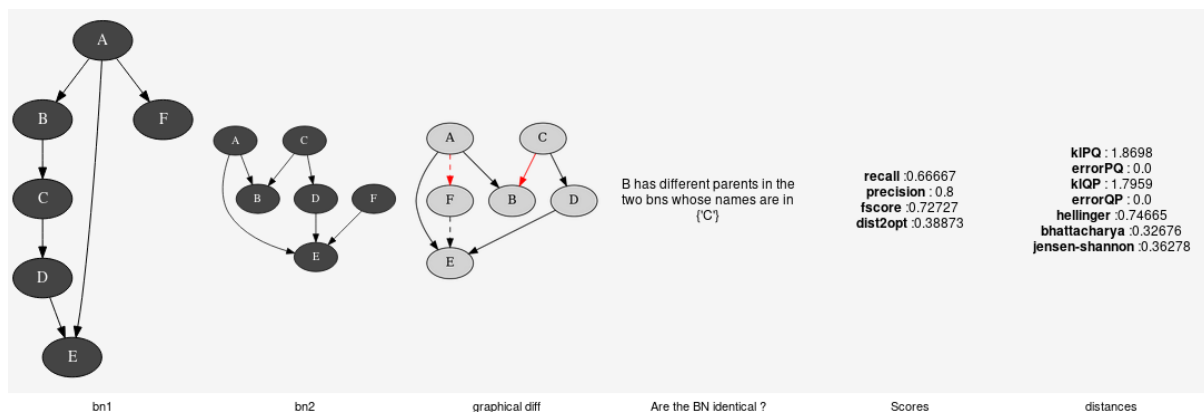
**Raises** **pyAgrum.IndexError** – If the graph does not contain the variable

**variableNodeMap**()

**Returns** the variable node map

**Return type** *pyAgrum.variableNodeMap*

## 4.2 Tools for Bayesian networks



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

### 4.2.1 Generation of database

**class** *pyAgrum.BNDatabaseGenerator*(*bn*)

*BNDatabaseGenerator* is used to easily generate databases from a *pyAgrum.BayesNet*.

**Parameters** *bn* (*pyAgrum.BayesNet* (page 58)) – the Bayesian network used to generate data.

**database**()

**drawSamples**(*nbSamples*)

**Parameters** *nbSamples* (*int*) –

**Return type** *float*

**log2likelihood**()

**Return type** *float*

**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

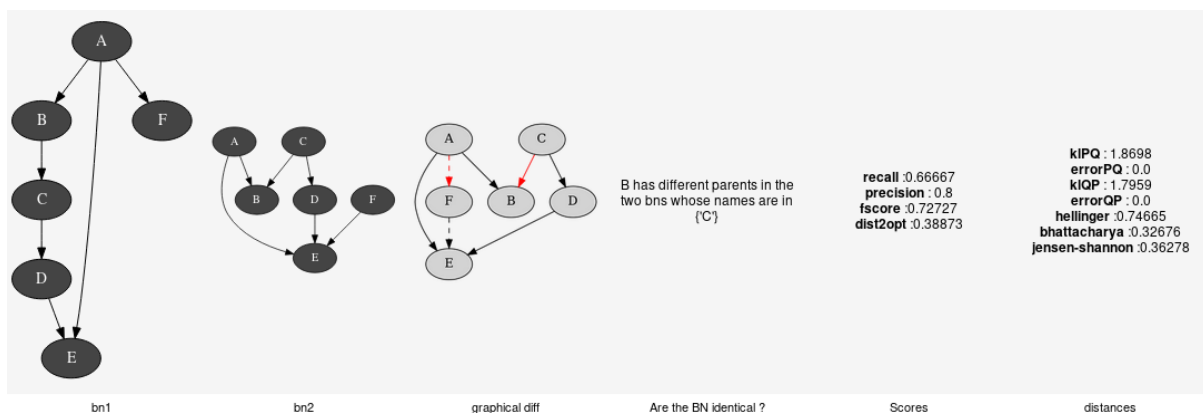
**toDatabaseTable(useLabels=True)**

**varOrder()**

**varOrderNames()**

**Return type** List[str]

## 4.2.2 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) –

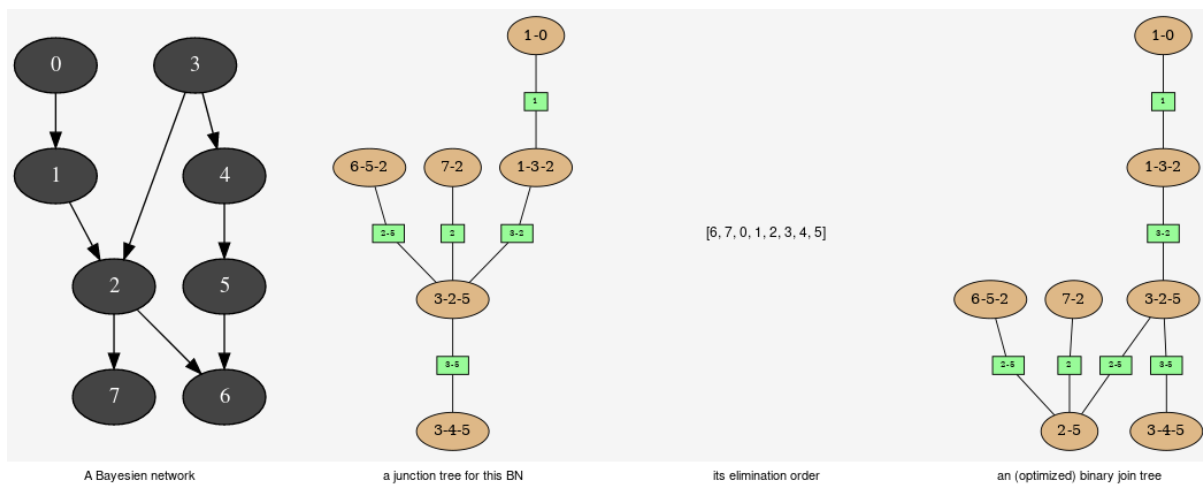
**Return type** None

**verbosity()**

**Returns** True if the verbosity is enabled

**Return type** bool

### 4.2.3 Explanation and analysis



This tools aimed to provide some different views on the Bayesian network in order to explore its qualitative and/or quantitive 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 heurisits 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 11)) – a undirected graph
- **dag** (`pyAgrum.DAG` (page 8)) – a dag
- **bn** (`pyAgrum.BayesNet` (page 58)) – a BayesianNetwork
- **partial\_order** (`List[List[int]]`) – a partial order among the nodeIDs

**Returns** the current binary joint tree

**Return type** *pyAgrum.CliqueGraph* (page 14)

**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 11)) – a undirected graph
- **dag** (*pyAgrum.DAG* (page 8)) – a dag
- **bn** (*pyAgrum.BayesNet* (page 58)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns** the current elimination order.

**Return type** *pyAgrum.CliqueGraph* (page 14)

**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 11)) – a undirected graph
- **dag** (*pyAgrum.DAG* (page 8)) – a dag
- **bn** (*pyAgrum.BayesNet* (page 58)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns** the current junction tree.

**Return type** *pyAgrum.CliqueGraph* (page 14)

**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 18)

**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 11)

**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

## 4.2.4 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 refered by the fragment

**Parameters** **bn** (*IBayesNet*) –

**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

**ancestors**(*norid*)

**Parameters** *norid* (object) –

**Return type** object

**arcs**()

**Returns** The list of arcs in the IBayesNet

**Return type** list

**checkConsistency**(\**args*)

If a variable is added to the fragment but not its parents, there is no CPT consistent for this variable. This function checks the consistency for a variable or 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 290) –
- **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 42)

**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 48)

**Raises** *pyAgrum.NotFound* (page 290) – 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 290) – 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 290) –
- **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 48)) – the Potential to install

**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – 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 48)) – the Potential (marginal) to install

**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – 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 290) – 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*) – The ids of the targets
- **list** (*list*) – The list of available variables

**Returns** The minimal set of variables**Return type** Set

**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 11)

**Parameters** **clear** (bool) –

**moralizedAncestralGraph**(*nodes*)

**Parameters** **nodes** (object) –

**Return type** *UndiGraph* (page 11)

**names**()

**Returns** The names of the graph variables

**Return type** list

**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

**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 58)

**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 289) – 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 290) – 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 290) – 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

## 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.

## 4.4 Exact Inference

### 4.4.1 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 288) – 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** – (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 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 288) – 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 48)

**evidenceJointImpact**(\**args*)

Create a pyAgrum.Potential for P(joint targets|evs) (for all instantiation of targets and evs)

**Parameters**

- **targets** – (int) a node Id
- **targets** – (str) a node name
- **evs** (*set*) – a set of nodes ids or names.

**Returns** a Potential for P(target|evs)

**Return type** *pyAgrum.Potential* (page 48)

**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

**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

**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 14)

**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 48)

**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 14)

**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 48)

**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

**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 288) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

#### 4.4.2 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 288) – 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** – (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 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 288) – 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.

<b>Warning:</b> 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 48)

**evidenceJointImpact**(\**args*)

Create a pyAgrum.Potential for P(joint targets|evs) (for all instantiation of targets and evs)

**Parameters**

- **targets** – (int) a node Id
- **targets** – (str) a node name
- **evs** (*set*) – a set of nodes ids or names.

**Returns** a Potential for P(target|evs)

**Return type** *pyAgrum.Potential* (page 48)

**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

**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

**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 14)

**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 48)

**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 14)

**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 48)

**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

**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 288) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

### 4.4.3 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 288) – 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** – (*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 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 288) – 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  $P(\text{target}|\text{evs})$  (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 48)

**evidenceJointImpact**(*targets, evs*)

Create a *pyAgrum.Potential* for  $P(\text{joint targets}|\text{evs})$  (for all instantiation of targets and evs)

**Parameters**

- **targets** (*object*) – (*int*) a node Id
- **targets** – (*str*) a node name
- **evs** (*set*) – a set of nodes ids or names.

**Returns** a Potential for  $P(\text{target}|\text{evs})$

**Return type** *pyAgrum.Potential* (page 48)

**Raises** **pyAgrum.Exception** – If some evidence entered into the Bayes net are incompatible (their joint proba = 0)

**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

**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 48)

**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 14)

**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 48)

**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

**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 288) – If one value is a vector of 0s
- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network

## 4.5 Approximated Inference

### 4.5.1 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 288) – 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** – (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 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 288) – 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  $P(\text{target}|\text{evs})$  (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 48)

**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 48)

**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 `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 288) – 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

## 4.5.2 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 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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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  $\text{eps} < 0$

**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

**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 288) – 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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 `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 288) – 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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 288) – 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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 288) – 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`

### 4.5.3 Loopy sampling

#### Loopy Gibbs Sampling

**class** `pyAgrum.LoopyGibbsSampling(bn)`

Class used for inferences using a loopy version of Gibbs sampling.

`LoopyGibbsSampling(bn) -> LoopyGibbsSampling`

**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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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  $\text{eps} < 0$

**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

**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 288) – 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)`

Class used for inferences using a loopy version of Monte Carlo sampling.

**LoopyMonteCarloSampling(bn) -> LoopyMonteCarloSampling**

**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 288) – 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`** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 288) – 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*)

Class used for inferences using a loopy version of weighted sampling.

**LoopyWeightedSampling(bn) -> LoopyWeightedSampling**

**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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 288) – 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*)

Class used for inferences using a loopy version of importance sampling.

**LoopyImportanceSampling(bn) -> LoopyImportanceSampling**

**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 288) – 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** – (*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 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 288) – 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 48)

**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 48)

**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 48)

**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 288) – 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



## 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 developping new learning algorithms (for instance, compute chi2 or conditionnl likelihood on the database, etc.).

**class** `pyAgrum.BNLearner(filename, inducedTypes=True)` → [BNLearner](#) (page 173)

### Parameters:

- **filename** (*str*) – the file to learn from
- **inducedTypes** (*Bool*) – whether `BNLearner` should try to automatically find the type of each variable

**BNLearner(filename,src) -> BNLearner**

### Parameters:

- **filename** (*str*) – the file to learn from
- **src** (*pyAgrum.BayesNet*) – the Bayesian network used to find those modalities

**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)**

The arc in parameters won't be added.

### Parameters

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

**Return type** None

**addMandatoryArc(\*args)**

Allow to add prior structural knowledge.

### Parameters

- **arc** ([pyAgrum.Arc](#) (page 3)) – an arc
- **head** – a variable's id (int)
- **tail** – a variable's id (int)

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

Raises [\*pyAgrum.InvalidDirectedCycle\*](#) (page 289) – If the added arc creates a directed cycle in the DAG

**Return type** None

**addPossibleEdge**(\*args)

**Return type** None

**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

**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)

Allow the arc to be added if necessary.

**Parameters**

- **arc** (*pyAgrum*) – an arc
- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)

**Return type** None

**eraseMandatoryArc**(\*args)

**Parameters**

- **arc** (*pyAgrum*) – an arc
- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)

**Return type** None

**erasePossibleEdge**(\*args)

Allow the 2 arcs to be added if necessary.

**Parameters**

- **arc** (*pyAgrum*) – an arc
- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)

**Return type** None

**hasMissingValues**()

Indicates whether there are missing values in the database.

**Returns** True if there are some missing values in the database.

**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\_names** (*str*) – a variable's name
- **var\_name** (*str*) –

**Returns** the column id corresponding to a variable name

**Return type** int

**Raises** [\*pyAgrum.MissingVariableInDatabase\*](#) – If a variable of the BN is not found in the database.

**latentVariables**(\*args)

**Warning:** 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 58)

**learnDAG()**

learn a structure from a file

**Returns** the learned DAG

**Return type** *pyAgrum.DAG* (page 8)

**learnMixedStructure()**

**Warning:** learner must be using 3off2 or MIIC algorithm

**Returns** the learned structure as an EssentialGraph

**Return type** *pyAgrum.EssentialGraph* (page 79)

**learnParameters(\*args)**

learns a BN (its parameters) when its structure is known.

**Parameters**

- **dag** (*pyAgrum.DAG* (page 8)) –
- **bn** (*pyAgrum.BayesNet* (page 58)) –
- **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 apriori (like K2 which has a 1-smoothing apriori), it is important to also take into account this implicit apriori for parameter learning. By default, if a score exists, we will learn parameters by taking into account the apriori specified by methods *useAprioriXXX ()* + the implicit apriori of the score, else we just take into account the apriori specified by *useAprioriXXX ()*

**Returns** the learned BayesNet

**Return type** *pyAgrum.BayesNet* (page 58)

**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)**

logLikelihood computes the log-likelihood for the columns in vars, given the columns in the list knowing (optional)

**Parameters**

- **vars** (*List[str]*) – the name of the columns of interest
- **knowing** (*List[str]*) – the (optional) list of names of conditioning columns

**Returns** the log-likelihood (base 2)

**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

**nameFromId(*id*)**

**Parameters** **id** (int) – a node id

**Returns** the variable's name

**Return type** str

**names()**

**Returns** the names of the variables in the database

**Return type** List[str]

**nbCols()**

Return the number of columns in the database

**Returns** the number of columns in the database

**Return type** int

**nbRows()**

Return the number of row in the database

**Returns** the number of rows in the database

**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

**setAprioriWeight**(weight)

Deprecated methods in BN Learner for pyAgrum>0.14.0

**setDatabaseWeight**(new\_weight)

Set the database weight which is given as an equivalent sample size.

**Parameters**

- **weight** (float) – the database weight
- **new\_weight** (float) –

**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

**setInitialDAG**(g)

**Parameters**

- **dag** ([pyAgrum.DAG](#) (page 8)) – an initial DAG structure
- **g** ([DAG](#) (page 8)) –

**Return type** None

**setMaxIndegree**(max\_indegree)

**Parameters** max\_indegree (int) –

**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

**setPeriodSize**(*p*)

**Parameters** **p** (*int*) – number of samples between 2 stopping

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

**Return type** None

**setPossibleSkeleton**(*skeleton*)

**Parameters** **skeleton** ([\*UndiGraph\*](#) (page 11)) –

**Return type** None

**setRecordWeight**(*i*, *weight*)

**Parameters**

- **i** (*int*) –
- **weight** (*float*) –

**Return type** None

**setSliceOrder**(\**args*)

Set a partial order on the nodes.

**Parameters** **l** (*list*) – a list of sequences (composed of ids of rows or string)

**Return type** None

**setVerbosity**(*v*)

**Parameters** **v** (*bool*) – verbosity

**Return type** None

**state**()

**Return type** object

**use3off2**()

Indicate that we wish to use 3off2.

**Return type** None

**useAprioriBDeu**(\**args*)

The BDeu apriori adds weight to all the cells of the counting tables. In other words, it adds weight rows in the database with equally probable values.

**Parameters** **weight** (*float*) – the apriori weight

**Return type** None

**useAprioriDirichlet**(\**args*)

**Return type** None

**useAprioriSmoothing**(\*args)

**Return type** None

**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** None

**useGreedyHillClimbing**()

**Return type** None

**useK2**(\*args)

Indicate to use the K2 algorithm (which needs a total ordering of the variables).

**Parameters** **order** (*list[int or str]*) – sequences of (ids or name)

**Return type** None

**useLocalSearchWithTabuList**(\*args)

Indicate that we wish to use a local search with tabu list

**Parameters**

- **tabu\_size** (*int*) – The size of the tabu list
- **nb\_decrease** (*int*) – The max number of changes decreasing the score consecutively that we allow to apply

**Return type** None

**useMDLCorrection**()

Indicate that we wish to use the MDL correction for 3off2 or MIIC

**Return type** None

**useMIIC**()

Indicate that we wish to use MIIC.

**Return type** None

**useNMLCorrection**()

Indicate that we wish to use the NML correction for 3off2 or MIIC

**Return type** None

**useNoApriori**()

**Return type** None

**useNoCorrection**()

Indicate that we wish to use the NoCorr correction for 3off2 or MIIC

**Return type** None

**useScoreAIC**()

**Return type** None

**useScoreBD**()

**Return type** None



**useScoreBDeu()**

**Return type** None

**useScoreBIC()**

**Return type** None

**useScoreK2()**

**Return type** None

**useScoreLog2Likelihood()**

**Return type** None

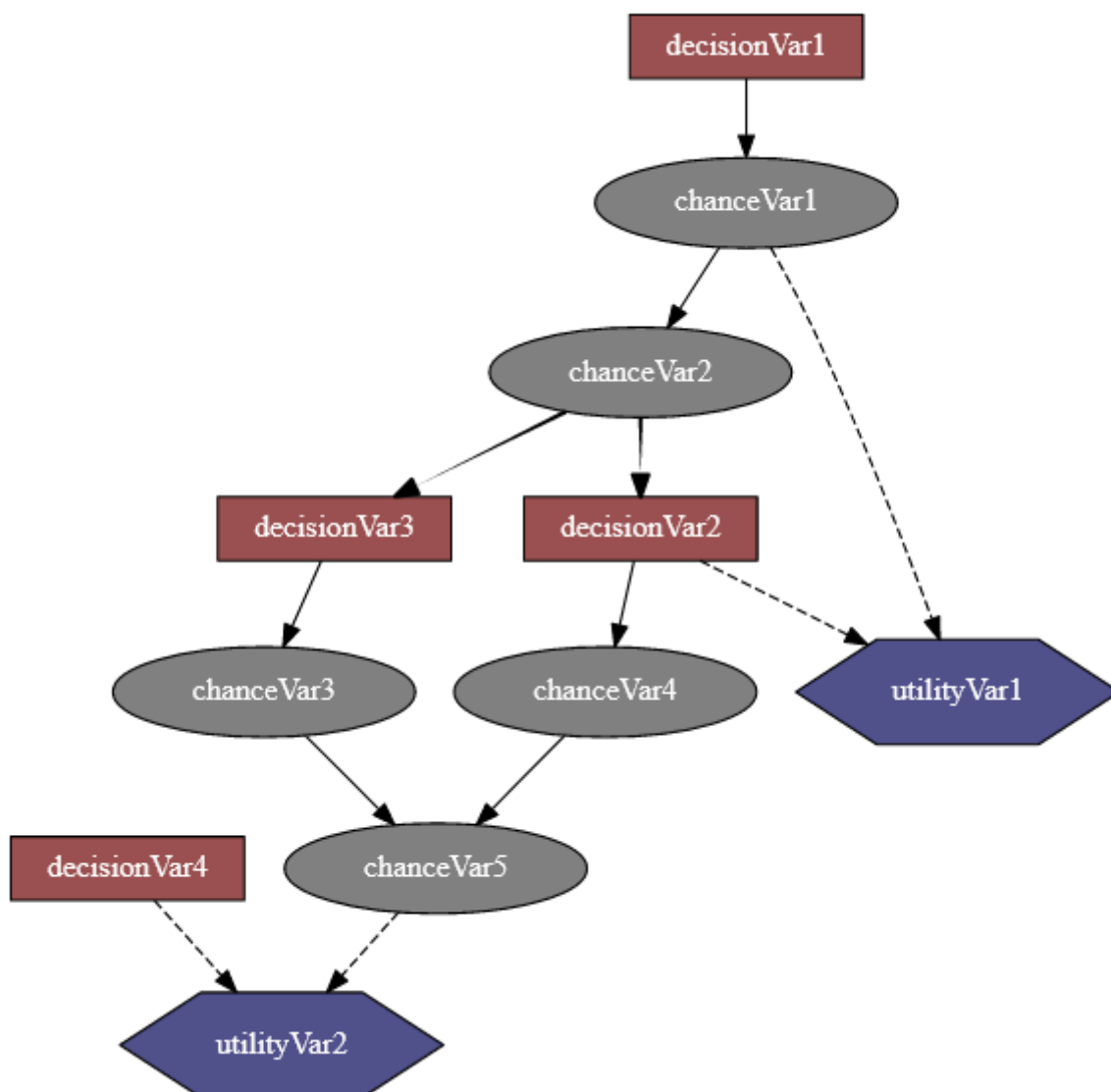
**verbosity()**

**Returns** True if the verbosity is enabled

**Return type** bool



## 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](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/InfluenceDiagram.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/InfluenceDiagram.ipynb.html>)

## Reference

### 5.1 Model

**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(variable, id=0)**

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

**addArc(\*args)**

Add an arc in the ID, and update diagram's potential nodes cpt if necessary.

**Parameters**

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

**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`

**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**(*variable*, *id=0*)

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

**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

**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**

- **id** (`int`) – the node Id
- **new\_name** (`str`) – the name of the variable

**Raises**

- [\*pyAgrum.DuplicateLabel\*](#) (page 287) – If this name already exists
- [\*pyAgrum.NotFound\*](#) (page 290) – If no nodes matches id.

**Return type** `None`

**children**(*norid*)

**Parameters**

- **id** (`int`) – the id of the parent
- **norid** (`object`) –

**Returns** the set of all the children

**Return type** `Set`

**clear**()

**Return type** `None`

**completeInstantiation**()

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

**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.BayesNet.

**Returns** The variable's CPT.

**Return type** [\*pyAgrum.Potential\*](#) (page 48)

**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – 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** (*pyAgrum.DiscreteVariable* (page 25)) – The 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.
- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

**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.

---

**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 184)



**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 290) – 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 288) – 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 11)

**Parameters** **clear** (bool) –

**moralizedAncestralGraph**(*nodes*)

**Parameters** **nodes** (object) –

**Return type** *UndiGraph* (page 11)

**names**()

**Returns** The names of the InfluenceDiagram variables

**Return type** list

**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**

- **id** – The id of the child node
- **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 289) – If this graph contains cycles

**Parameters** **clear** (bool) –

**utility(\*args)**

**Parameters** **varId** (*int*) – the tested node id.

**Returns** the utility table of the node

**Return type** [\*pyAgrum.Potential\*](#) (page 48)

**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 variabe given it's node id

**Return type** [\*pyAgrum.DiscreteVariable\*](#) (page 25)

**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – If no variable's id matches the parameter

**variableFromName(*name*)**

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

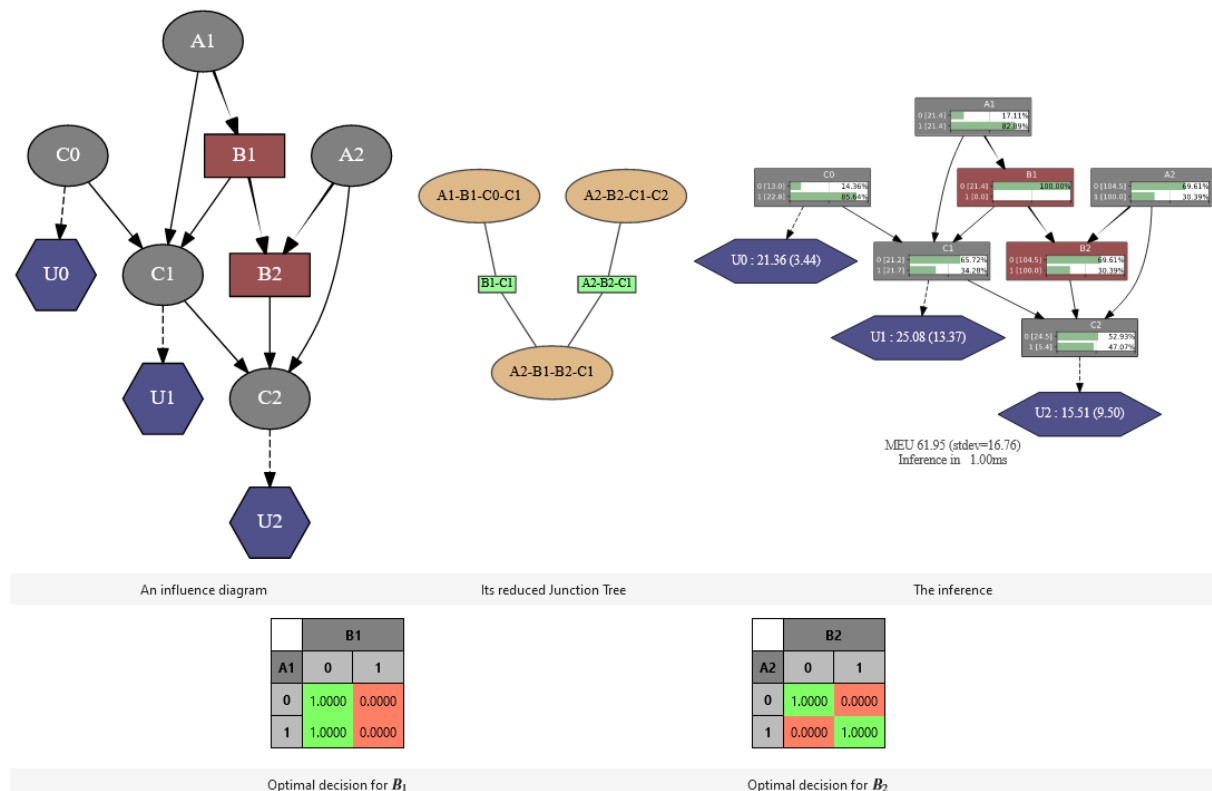
**Returns** the variable

**Return type** [\*pyAgrum.DiscreteVariable\*](#) (page 25)

**Raises** [\*pyAgrum.IndexError\*](#) – If the InfluenceDiagram does not contain the variable

**variableNodeMap()**

## 5.2 Inference



**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 184)) –

**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 48)) – 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 184)

**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 48)

**posterior(\*args)**

Return type [`Potential`](#) (page 48)

**posteriorUtility(\*args)**

Return type [`Potential`](#) (page 48)

**reducedGraph()**

Return type [`DAG`](#) (page 8)

**reducedLIMID()**

Return type [`InfluenceDiagram`](#) (page 184)

**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 288) – 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 288) – If one value is a vector of 0s
- *pyAgrum.UndefinedElement* (page 291) – If one node does not belong to the Bayesian network





## 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](https://en.wikipedia.org/wiki/Credal_network))).

### Tutorial

- [Tutorial on Credal Networks](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/credalNetworks.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/credalNetworks.ipynb.html>)

### Reference

## 6.1 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 58)) – The path to a BayesNet or the BN itself which contains lower probabilities.
- **src\_max\_den** (*str* or [pyAgrum.BayesNet](#) (page 58)) – 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 if provided with counts, to False if two BayesNet 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 choosen NodeId

**Returns** the type of the choosen node in the (up-to-date) CredalNet `__current_bn` if any, `__src_bn` otherwise.

**Return type** *pyAgrum.CredalNet* (page 197)

**current\_bn()**

**Returns** Returs a constant reference to the actual BayesNet (used as a DAG, it's CPTs does not matter).

**Return type** *pyAgrum.BayesNet* (page 58)

**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 42)) – The Instantiation
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

<b>Warning:</b> You need to call <code>intervalToCredal</code> 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.

bnnet accessors / shortcuts.

**Parameters** *id* (*int*) – the id of the node we want an instantiation from

**Returns** the instantiation

**Return type** *pyAgrum.Instantiation* (page 42)

**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 197)

**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 42)) – 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 58)

## 6.2 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 197)) –

**CN**()

**Return type** [CredalNet](#) (page 197)

**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 `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`

**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 197)) –

**CN**()

**Return type** *CredalNet* (page 197)

**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  $\text{eps} < 0$

**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  $\text{timeout} \leq 0.0$

**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

**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

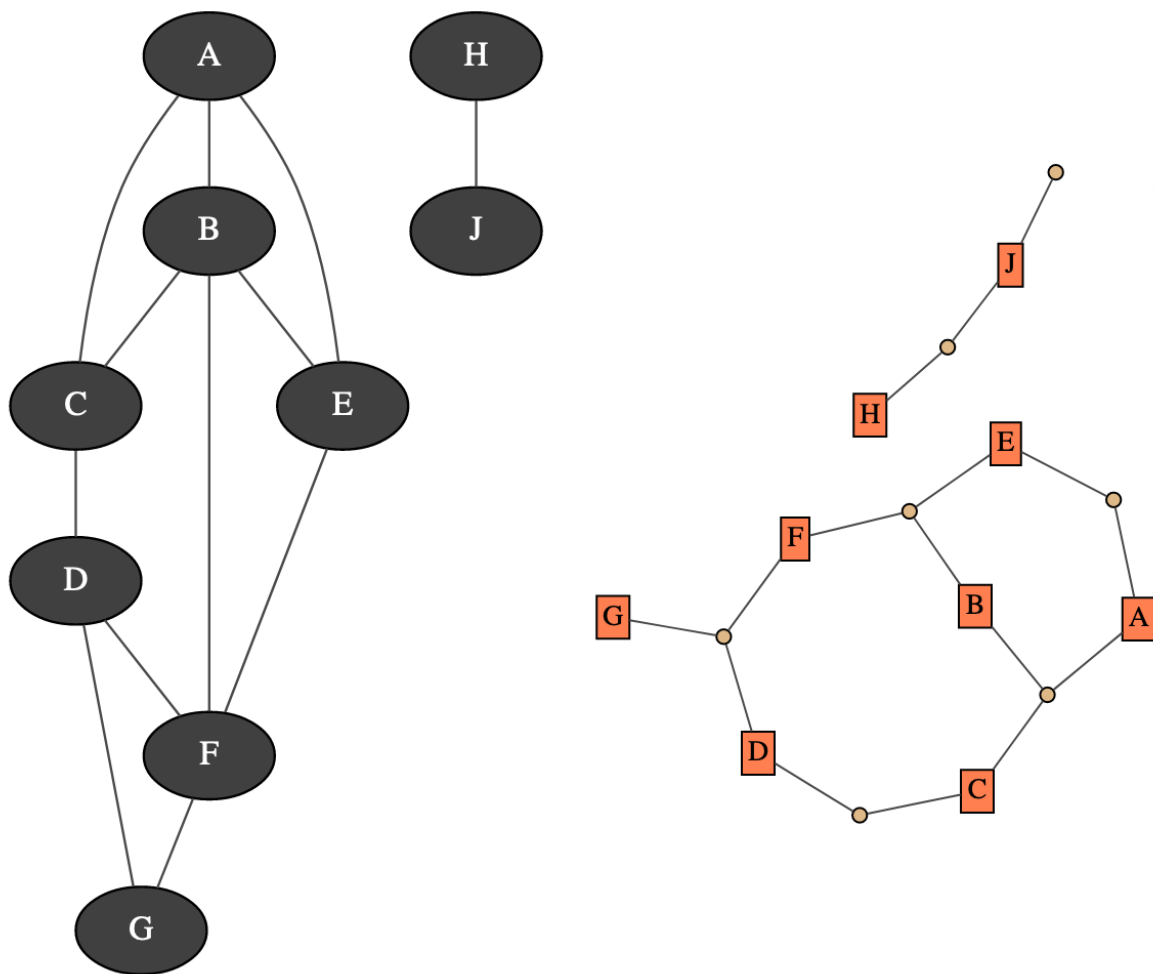
**verbosity()**

**Returns** True if the verbosity is enabled

**Return type** bool



## 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](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/current/notebooks/MarkovNetwork.ipynb) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/current/notebooks/MarkovNetwork.ipynb>)

## Reference

### 7.1 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 48)

**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

**beginTopologyTransformation()**



**Return type** None

**changeVariableLabel**(\*args)

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

**Parameters**

- **id** (*int*) – the id of the node
- **name** (*str*) – the name of the variable
- **old\_label** (*str*) – the new label
- **new\_label** (*str*) – the new label

**Raises** [\*pyAgrum.NotFound\*](#) (page 290) – 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**

- **new\_name** (*str*) – the new name of the variable
- **NodeId** (*int*) – the id of the node
- **name** (*str*) – the name of the variable

**Raises**

- [\*pyAgrum.DuplicateLabel\*](#) (page 287) – If new\_name is already used in this MarkovNet.
- [\*pyAgrum.NotFound\*](#) (page 290) – If no variable matches id.

**Return type** None

**clear()**

Clear the whole MarkovNet

**Return type** None

**completeInstantiation()**

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

**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 212)

**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**

- **id** (*int*) – The variable's id to remove.
- **name** (*str*) – The variable's name to remove.
- **var** (*pyAgrum.DiscreteVariable* (page 25)) – A reference on the 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**

- **VarId** (*Set[int]*) – A variable's id in the pyAgrum.MarkovNet.
- **name** (*Set[str]*) – A variable's name in the pyAgrum.MarkovNet.

**Returns** The factor of the set of nodes.

**Return type** *pyAgrum.Potential* (page 48)

**Raises** *pyAgrum.NotFound* (page 290) – 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 212)

**static fromBN**(*bn*)

**Parameters** *bn* (*BayesNet* (page 58)) –

**Return type** *MarkovNet* (page 212)

**generateFactor**(*vars*)

Randomly generate factor parameters for a given factor in a given structure.

### Parameters

- **node** (*int*) – The variable's id.
- **name** (*str*) – The variable's name.
- **vars** (*List[int]*) –

**Return type** *None*

**generateFactors**()

Randomly generates factors parameters for a given structure.

**Return type** *None*

**graph**()

**Return type** *UndiGraph* (page 11)

**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 288) – 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()**

**Return type** *object*

**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**()

## 7.2 Inference

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.

## 7.2.1 Shafer Shenoy Inference

**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 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** 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** 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 288) – 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** – (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 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 288) – 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 48)

**evidenceJointImpact**(*\*args*)

Create a `pyAgrum.Potential` for  $P(\text{joint targets}|\text{evs})$  (for all instantiation of targets and evs)

**Parameters**

- **targets** – (int) a node Id
- **targets** – (str) a node name
- **evs** (*set*) – a set of nodes ids or names.

**Returns** a Potential for  $P(\text{target}|\text{evs})$

**Return type** `pyAgrum.Potential` (page 48)

**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

**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

**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 14)

**jointMutualInformation**(targets)

**Parameters** **targets** (object) –

**Return type** float

**jointPosterior**(targets)

Compute the joint posterior of a set of nodes.

**Parameters** `targets` (object) – the list of nodes whose posterior joint probability is wanted

**Returns** a const ref to the posterior joint probability of the set of nodes.

**Return type** *pyAgrum.Potential* (page 48)

**Raises** *pyAgrum.UndefinedElement* (page 291) – If an element of nodes is not in targets

**Warning:**

- The order of the variables given by the targets here or when the jointTarget is declared can not be assumed to be used by the Potential.
- The nodes with hard evidence are removed from the targets.

### `jointTargets()`

**Returns** the list of target sets

**Return type** list

### `junctionTree()`

**Returns** the current junction tree

**Return type** *pyAgrum.CliqueGraph* (page 14)

### `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 48)

**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 side of the node
- *pyAgrum.FatalError* (page 288) – If one value is a vector of 0s
- *pyAgrum.UndefinedElement* (page 291) – If one node does not belong to the Bayesian network

**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 288) – If one value is a vector of 0s

- [\*pyAgrum.UndefinedElement\*](#) (page 291) – If one node does not belong to the Bayesian network



## 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 48)

**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 interface 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 288) – If file not found

**Return type** None

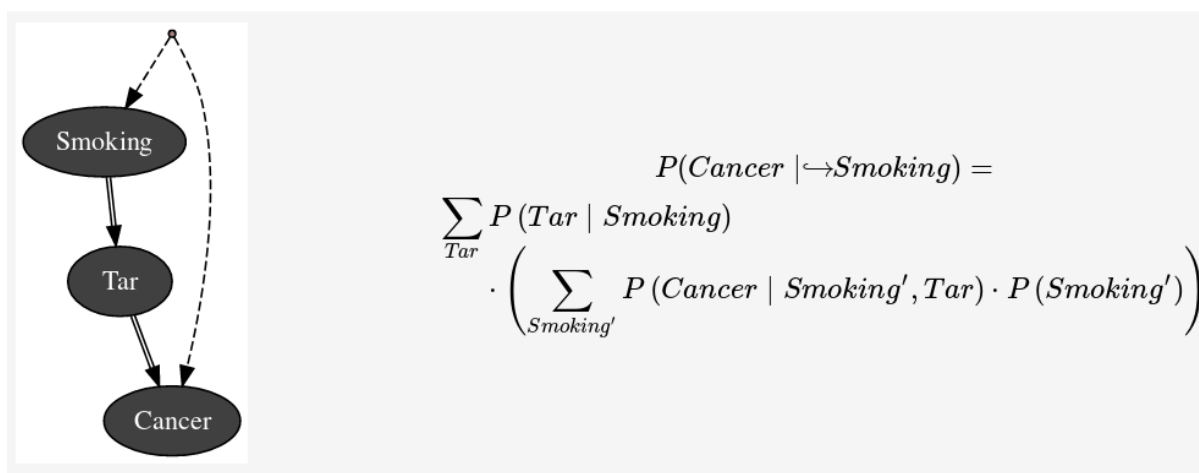
**types**()

**Returns** the list of the custom types in the PRM

**Return type** list



## 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"]=")"
```

---

### Tutorial

- [Notebooks on causality in pyAgrum](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Tobacco.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Tobacco.ipynb.html>)
- Some [implemented examples](https://webia.lip6.fr/~phw/aGrUM/BookOfWhy/) (<https://webia.lip6.fr/~phw/aGrUM/BookOfWhy/>) from the [book of Why](http://bayes.cs.ucla.edu/WHY/) (<http://bayes.cs.ucla.edu/WHY/>) from Judea Pearl and Dana Mackenzie.

### Reference

## 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** (*BayesNet* (page 58)) – a observational Bayesian network
- **latentVarsDescriptor** (Optional[List[Tuple[str, Tuple[str, str]]]]) – 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** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the first node
- **y** (Union[NewType()(NodeId, 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 effet

### Parameters

- **cause** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the cause
- **effect** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the effect
- **withNames** (bool) – does the function return the set of NodeId or the set of name ?

**Return type** Union[None, Set[str], Set[NewType()(NodeId, int)]]

**Returns** None if no backdoor has been found. Otherwise the set of NodeId or names of the backdoor.

**causalBN**()

**Return type** *BayesNet* (page 58)

**Returns** the causal Bayesian network

**Warning** do not infer any computations in this model. It is strictly a structural model

**children**(*x*)

**Parameters** *x* (Union[NewType()(NodeId, int), str]) – the node

**Return type** Set[NewType()(NodeId, int)]

**Returns**

**eraseCausalArc**(*x*, *y*)

Erase the arc *x*->*y*

**Parameters**

- **x** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the first node
- **y** (Union[NewType()(NodeId, 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** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the first node
- **y** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the second node

**Return type** bool

**Returns** True if the arc exists.

**frontDoor**(*cause*, *effect*, *withNames=True*)

Check if a frontdoor exists between cause and effet

**Parameters**

- **cause** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the cause
- **effect** (Union[NewType()(NodeId, int), str]) – the nodeId or the name of the effect
- **withNames** (bool) – does the function return the set of NodeId or the set of name ?

**Return type** Union[None, Set[str], Set[NewType()(NodeId, int)]]

**Returns** None if no frontdoor has been found. Otherwise the set of NodeId or names of the frontdoor.

**idFromName**(*name*)

**Parameters** *name* (str) – the name of the variable

**Return type** NewType()(NodeId, int)

**Returns** the id of the variable

**latentVariablesIds**()

**Return type** Set[NewType()(NodeId, int)]

**Returns** the set of ids of latent variables in the causal model

**names**()

**Return type** Dict[NewType()(NodeId, int), str]

**Returns** the map NodeId,Name

**nodes()**

**Return type** Set[NewType()(NodeId, int)]

**Returns** the set of nodes

**observationalBN()**

**Return type** [BayesNet](#) (page 58)

**Returns** the observational Bayesian network

**parents(x)**

From a NodeId, returns its parent (as a set of NodeId)

**Parameters** **x** (Union[NewType()(NodeId, int), str]) – the node

**Return type** Set[NewType()(NodeId, int)]

**Returns**

**toDot()**

Create a dot representation of the causal model

**Return type** str

**Returns** the dot representation in a string

## 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 234)) – the causal model
- **root** ([ASTtree](#) (page 239)) – the syntax tree as the root ASTtree
- **on** (Union[str, Set[str]]) – the variable or the set of variables of interest
- **doing** (Union[str, Set[str]]) – the intervention variables
- **knowing** (Optional[Set[str]]) – the observation variables

**property** cm: [pyAgrum.causal.\\_CausalModel.CausalModel](#) (page 234)

return: the causal model

**Return type** [CausalModel](#) (page 234)

**copy()**

Copy theAST. Note that the causal model is just referenced. The tree is copied.

**Return type** [CausalFormula](#) (page 236)

**Returns** the new CausalFormula



**eval()**

Compute the Potential from the CausalFormula over vars using cond as value for others variables

**Return type** *Potential* (page 48)

**Returns**

**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** (Optional[Dict[str, str]]) – the values to add in the query representation

**Return type** str

**Returns** the string representing the causal query for this CausalFormula

**property root: pyAgrum.causal.\_doAST.ASTtree (page 239)**

return: ASTtree root of the CausalFormula tree

**Return type** *ASTtree* (page 239)

**toLatex()**

**Return type** str

**Returns** a LaTeX representation of the CausalFormula

## 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 234)) – causal model
- **on** (Union[str, Set[str]]) – variable name or variable names set
- **doing** (Union[str, Set[str]]) – variable name or variable names set
- **knowing** (Optional[Set[str]]) – variable names set
- **values** (Optional[Dict[str, int]]) – Dictionary

**Return type** Tuple[*CausalFormula* (page 236), *Potential* (page 48), str]

**Returns** the CausalFormula, the computation, the explanation

**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**

- **on** (Set[str]) – the variables of interest

- **cm** ([CausalModel](#) (page 234)) – the causal model
- **doing** (Set[str]) – the interventions
- **knowing** (Optional[Set[str]]) – the observations

**Return type** [CausalFormula](#) (page 236)

**Returns** the CausalFormula for computing this causal impact

`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 234)) – 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 239)]) – The ASTtree representing the calculus in construction

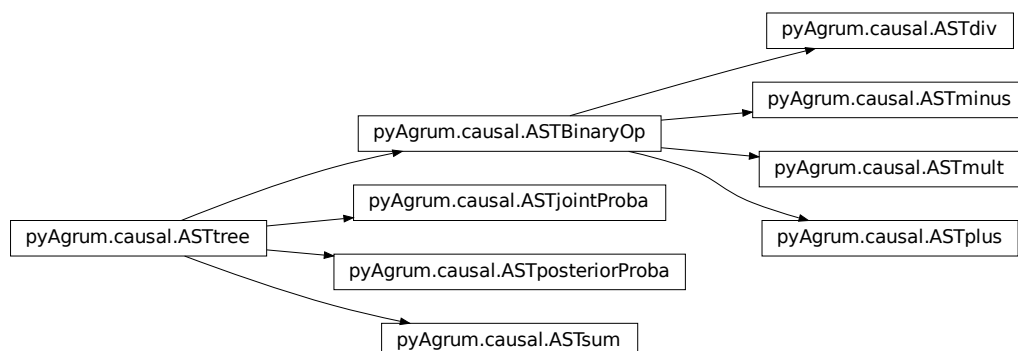
**Return type** [ASTtree](#) (page 239)

**Returns** the ASTtree representing the calculus

## 9.4 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 239).



### 9.4.1 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**

- **type** – the type of the node (will be specified in concrete children classes).
- **typ** (str) –

**copy()**

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new causal tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Optional[Dict[str, int]]) –

**property type:** str

return: 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**

- **type** – the type of the node (will be specified in concrete children classes)
- **op1** ([ASTtree](#) (page 239)) – left operand
- **op2** ([ASTtree](#) (page 239)) – right operand
- **typ** (str) –

**copy()**

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new causal tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Internal virtual function to create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**property op1:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the left operand

**Return type** [ASTtree](#) (page 239)

**property op2:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the right operand

**Return type** [ASTtree](#) (page 239)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Optional[Dict[str, int]]) –

**property type:** str

return: the type of the node

**Return type** str

## 9.4.2 Basic Binary Operations

**class** `pyAgrum.causal.ASTplus(op1, op2)`

Represents the sum of 2 `causal.ASTtree`

**Parameters**

- **op1** ([ASTtree](#) (page 239)) – first operand
- **op2** ([ASTtree](#) (page 239)) – second operand

**copy**()

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**property op1**: [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the left operand

**Return type** [ASTtree](#) (page 239)

**property op2**: [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the right operand

**Return type** [ASTtree](#) (page 239)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Optional[Dict[str, int]]) –

**property type**: str

return: the type of the node

**Return type** str

**class** [pyAgrum.causal.ASTminus](#)(*op1, op2*)

Represents the substraction of 2 [causal.ASTtree](#)

**Parameters**

- **op1** ([ASTtree](#) (page 239)) – first operand
- **op2** ([ASTtree](#) (page 239)) – second operand

**copy**()

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**property op1:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the left operand

**Return type** [ASTtree](#) (page 239)

**property op2:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the right operand

**Return type** [ASTtree](#) (page 239)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Optional[Dict[str, int]]) –

**property type:** str

return: 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 239)) – first operand
- **op2** ([ASTtree](#) (page 239)) – second operand

**copy**()

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**property op1:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the left operand

**Return type** [ASTtree](#) (page 239)

**property op2:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the right operand

**Return type** [ASTtree](#) (page 239)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Optional[Dict[str, int]]) –

**property type:** str

return: 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 239)) – first operand
- **op2** ([ASTtree](#) (page 239)) – second operand

**copy**()

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**property op1:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the left operand

**Return type** [ASTtree](#) (page 239)

**property op2:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the right operand

**Return type** [ASTtree](#) (page 239)

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Optional[Dict[str, int]]) –

**property type:** str

return: the type of the node

**Return type** str

### 9.4.3 Complex operations

**class** pyAgrum.causal.ASTsum(*var, term*)

Represents a sum over a variable of a causal.ASTtree.

**Parameters**

- **var** (List[str]) – name of the variable
- **term** ([ASTtree](#) (page 239)) – the tree to be evaluated

**copy**()

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of the sum

**Parameters** **contextual\_bn** ([BayesNet](#) (page 58)) – BN where to infer

**Return type** [Potential](#) (page 48)

**Returns** the value of the sum

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** **nameOccur** (Dict[str, int]) –

**property term:** [pyAgrum.causal.\\_doAST.ASTtree](#) (page 239)

return: the ASTtree of the expression inside the sum

**Return type** [ASTtree](#) (page 239)

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str



**Returns** the LaTeX string

**Parameters** `nameOccur` (Optional[Dict[str, int]]) –

**property type:** `str`

return: 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

**Return type** `ASTtree` (page 239)

**Returns** the new `CausalFormula` tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: `BayesNet` (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: `Potential` (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree :rtype: `str` :return: the LaTeX string

**Parameters** `nameOccur` (Dict[str, int]) –

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** `str`

**Returns** the LaTeX string

**Parameters** `nameOccur` (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** `str`

**Returns** the LaTeX string

**Parameters** `nameOccur` (Optional[Dict[str, int]]) –

**property type:** `str`

return: the type of the node

**Return type** `str`

**property varNames:** `Set[str]`

return: 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** (`BayesNet` (page 58)) – the `pyAgrum.pyAgrum.BayesNet`
- **varset** (Set[str]) – a set of variable names (in the BN)
- **knw** (Set[str]) – a set of variable names (in the BN)

**property bn:** `pyAgrum.pyAgrum.BayesNet` (page 58)

return: bn in  $P_{bn}(vars|knw)$

**Return type** [BayesNet](#) (page 58)

**copy()**

Copy an CausalFormula tree

**Return type** [ASTtree](#) (page 239)

**Returns** the new CausalFormula tree

**eval**(*contextual\_bn*)

Evaluation of a AST tree from inside a BN :type *contextual\_bn*: [BayesNet](#) (page 58) :param *contextual\_bn*: the BN in which will be done the computations :rtype: [Potential](#) (page 48) :return: the resulting Potential

**fastToLatex**(*nameOccur*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**property knw:** Set[str]

return: knw in  $P_{bn}(vars|knw)$

**Return type** Set[str]

**protectToLatex**(*nameOccur*)

Create a protected LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Dict[str, int]) –

**toLatex**(*nameOccur=None*)

Create a LaTeX representation of a ASTtree

**Return type** str

**Returns** the LaTeX string

**Parameters** *nameOccur* (Optional[Dict[str, int]]) –

**property type:** str

return: the type of the node

**Return type** str

**property vars:** Set[str]

return: vars in  $P_{bn}(vars|knw)$

**Return type** Set[str]

## 9.5 Exceptions

**class** pyAgrum.causal.HedgeException(*msg, observables, gs*)

Represents an hedge exception for a causal query

**Parameters**

- **msg** (str) – str
- **observables** (Set[str]) – NameSet
- **gs** – ???

**args**

```
class pyAgrum.causal.UnidentifiableException(msg)
    Represents an unidentifiability for a causal query

    args
```

## 9.6 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 :type
    model: CausalModel (page 234) :param model: the causal model :type on: Union[str, Set[str]] :param
    on: the impacted variable(s) :type doing: Union[str, Set[str]] :param doing: the variable(s) of inter-
    vention :type knowing: Optional[Set[str]] :param knowing: the variable(s) of evidence :param values :
    values for certain variables
```

**Return type** Tuple[str, [Potential](#) (page 48), str]

**Returns** a triplet (CausalFormula representation (string), pyAgrum.Potential, explanation)

**Parameters values** (Optional[Dict[str, int]]) –

```
pyAgrum.causal.notebook.getCausalModel(cm, size=None)
    return a HTML representing the causal model :type cm: CausalModel (page 234) :param cm: the causal
    model :param size: passd :param vals: :rtype: str :return:
```

```
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 :type
    model: CausalModel (page 234) :param model: the causal model :type on: Union[str, Set[str]] :param
    on: the impacted variable(s) :type doing: Union[str, Set[str]] :param doing: the variable(s) of inter-
    vention :type knowing: Optional[Set[str]] :param knowing: the variable(s) of evidence :param values :
    values for certain variables
```

**Parameters values** (Optional[Dict[str, int]]) –

```
pyAgrum.causal.notebook.showCausalModel(cm, size='4')
    Shows a graphviz svg representation of the causal DAG d
```

**Parameters**

- **cm** ([CausalModel](#) (page 234)) –
- **size** (str) –



## 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).

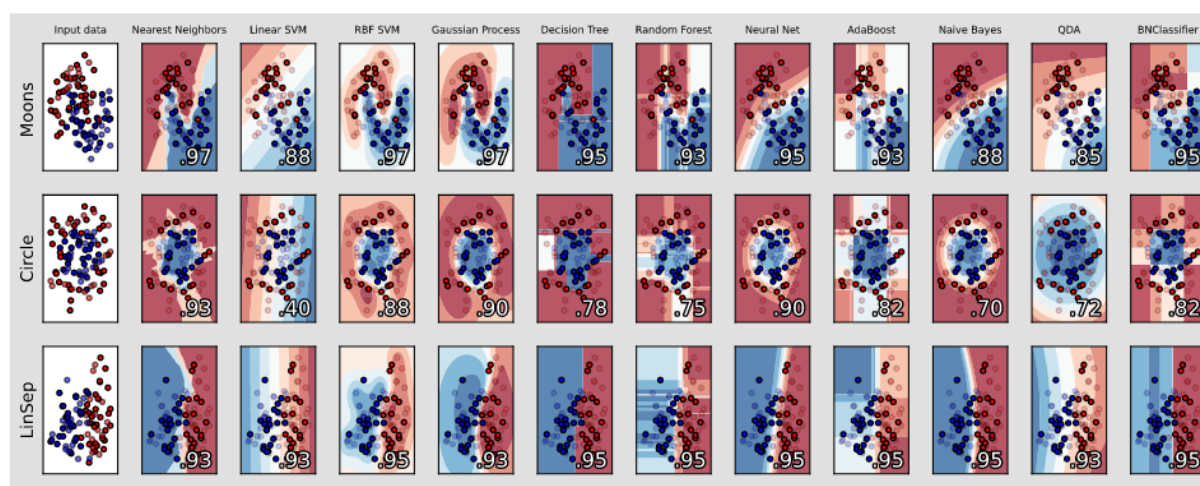


Fig. 1: An example from scikit-learn ([https://scikit-learn.org/stable/auto\\_examples/classification/plot\\_classifier\\_comparison.html](https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html)) where a last column with a BNClassifier has been added flawlessly (see [this notebook](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html>)).

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-like classifiers in pyAgrum](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html>), the [integration in scikit-learn codes](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CompareClassifiersWithSklern.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CompareClassifiersWithSklern.ipynb.html>) and, as an example, [cross-validation with scikit-learn](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CrossValidation.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CrossValidation.ipynb.html>)
- An [example from Kaggle](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/KaggleTitanic.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/KaggleTitanic.ipynb.html>),
- Notebook on [Discretizers in pyAgrum](https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Discretizer.ipynb.html) (<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Discretizer.ipynb.html>) useful for scikit-learn-like classifiers.

### Reference

## 10.1 Classifier using Bayesian networks

```
class pyAgrum.skbn.BNClassifier(learningMethod='GHC', aPriori=None, scoringType='BIC',  
                               constraints=None, aPrioriWeight=1, possibleSkeleton=None,  
                               DirichletCsv=None, discretizationStrategy='quantile',  
                               discretizationNbBins=5, discretizationThreshold=25, usePR=False,  
                               significant_digit=10)
```

Represents a (scikit-learn compliant) classifier wich 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 designated Tabu list searching

**aPriori: str** A string designating the type of a priori smoothing 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 aPriori 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.

**aPrioriWeight: double** The weight used for a priori smoothing.

**possibleSkeleton: pyagrundigraph** 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 aPriori 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 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 Precision-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*)

**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:**

**X: pandas.dataframe** Matrix containing the data

**y: pandas.dataframe** Column-vector containing the class for each data vector in X

Reads the data from a csv file and separates it into a X matrix and a y column vector.

**fit**(*X=None*, *y=None*, *filename=None*, *targetName=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

**filename: str** 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

**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.

**returns:** void

Fits the model to the training data provided. The two possible uses of this function are fit(X,y) and fit(filename, 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: {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*)

**parameters:**

**X: {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:**

**y: array-like of shape (n\_samples,)** Predicted probability for each classes

Predicts the probability of classes for each row of input data, with bn's Markov Blanket

**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:**

**csv\_name** [str] a csv filename

**save\_fig** [bool] whether the graph should be saved

**show\_progress** [bool] indicates if the resulting curve must be printed

## 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 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 example if you have entered 0.95, the variable will be treated as continuous 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:**

**clearDiscretizationParameters:** **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 continuous variables and the total number of bins created by this discretizer to 0. If `clearDiscretizationParameters` is True, also clears 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*)

**parameters:**

**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*)

**parameters:**

**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 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, methode=None, numberOfBins=None*)

**parameters:**

**variableName: str** the name of the variable you want to set the discretization parameters of. Set to None to set the new default for this BNClassifier.

**methode: 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'

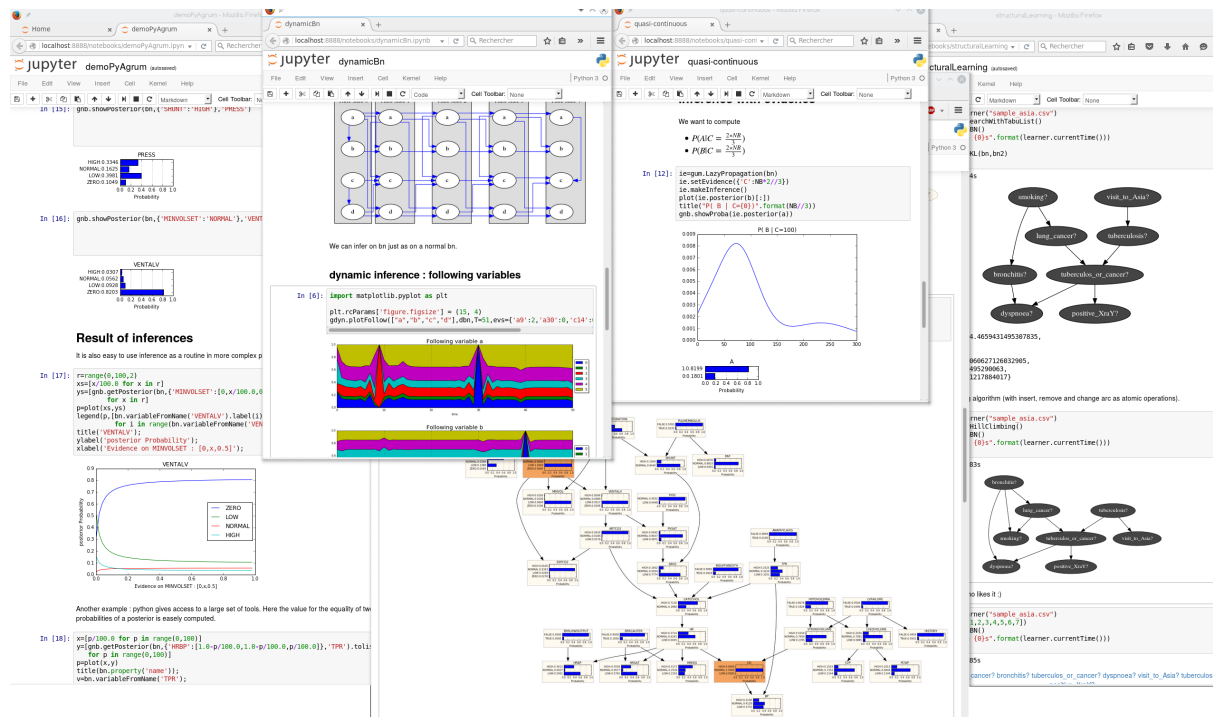
**numberOfBins:** sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. if the method used is NML, this parameter sets the maximum number of bins up to which 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

# PYAGRUM.LIB.NOTEBOOK

`pyAgrum.lib.notebook` aims to facilitate the use of `pyAgrum` with jupyter notebook (or lab).



## 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 mark “!” (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("* witht '!')
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 '!'



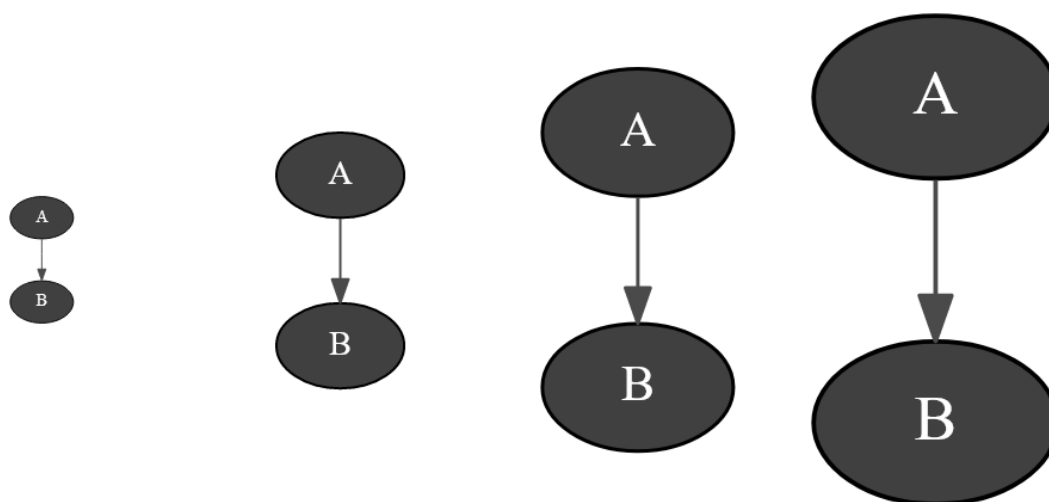
size="1"

size="2"

size="3"

size="4"

\* witht '!'



size="1!"

size="2!"

size="3!"

size="4!"

```

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** (*dictionary*) – 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** (*dictionary*) – 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

**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 representation of the 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

## 11.2 Visualization of Potentials

`pyAgrum.lib.notebook.showProba(p, scale=1.0)`  
Show a mono-dim Potential

**Parameters**

- **p** – the mono-dim Potential
- **scale** – the scale (zoom)

`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 get
- **digits** (*int*) – number of digits to show

- **varnames** (*list of strings*) – the aliases for variables name in the table

**Param** boolean withColors : bgcolor for proba cells or not

**Returns** the HTML string

`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 get
- **digits** (*int*) – number of digits to show
- **varnames** (*list of strings*) – the aliases for variables name in the table

**Param** boolean withColors : bgcolor for proba cells or not

**Returns** the display of the potential

## 11.3 Visualization of graphs

`pyAgrum.lib.notebook.getDot(dotstring, size=None)`  
get a dot string as a HTML string

**Parameters**

- **dotstring** – dot string
- **size** – size of the rendered graph
- **format** – render as “png” or “svg”
- **bg** – color for background

**Returns** the HTML representation of the graph

`pyAgrum.lib.notebook.showDot(dotstring, size=None)`  
show a dot string as a graph

**Parameters**

- **dotstring** – dot string
- **size** – size of the rendered graph

**Returns** the representation of the graph

`pyAgrum.lib.notebook.getGraph(gr, size=None)`  
get a HTML string representation of pydot graph

**Parameters**

- **gr** – pydot graph
- **size** – size of the rendered graph
- **format** – render as “png” or “svg”

**Returns** 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 graph
- **size** – size of the rendered graph

**Returns** the representation of the graph

## 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

## 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)

## 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, <<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/colouringAndExportingBNs.ipynb.html>>

```
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})
```

### 12.1 Visualization of models and inference

`pyAgrum.lib.image.export(model, filename, **kwargs)`

export the graphical representation of the model in filename (png, pdf, etc.)

#### Parameters

- **model** (*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'])

**Warning:** Model can also just possess a method *toDot()* or even be a simple string in dot syntax.

`pyAgrum.lib.image.exportInference(model, filename, **kwargs)`

the graphical representation of an inference in a notebook

#### Parameters

- **model** (*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'])
- **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** (*dictionary*) – 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.EXPLAIN

The purpose of `pyAgrum.lib.explain` is to give tools to explain and interpret the structure and parameters of a Bayesian network.

### 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

### 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** – the BN
- **evs** – map of evidence
- **size** – size of the graph
- **cmap** – colour map used

**Returns** the HTML string

`pyAgrum.lib.explain.showInformation`(*bn*, *evs=None*, *target=None*, *size=None*,  
*cmap=<matplotlib.colors.LinearSegmentedColormap object>*)

show a bn annotated with results from inference : entropy and mutual information

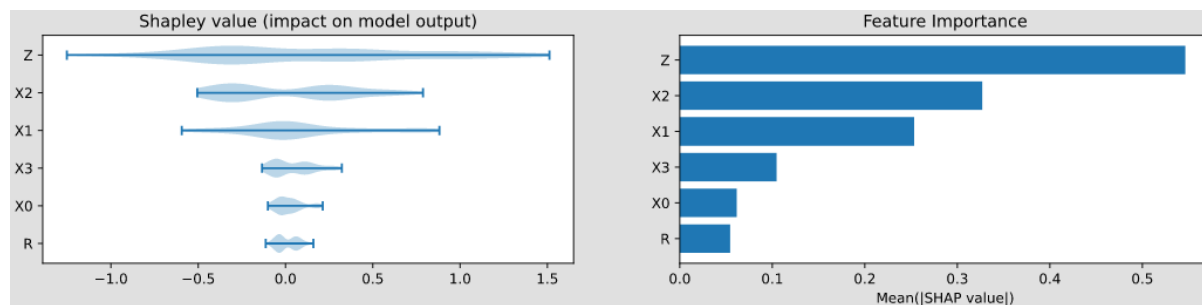
#### Parameters

- **bn** – the BN

- **evs** – map of evidence
- **target** – (optional) the name or id of the target variable
- **size** – size of the graph
- **cmap** – colour map used

**Returns** the graph

## 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]`

<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.



**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

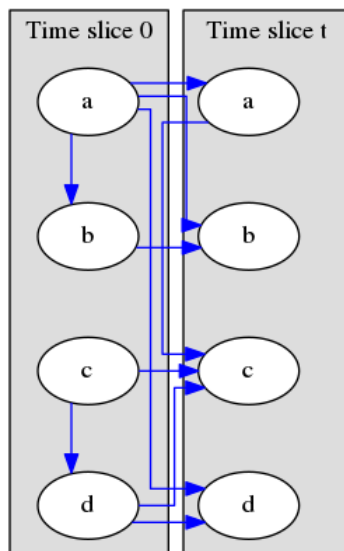


## PYAGRUM.LIB.DYNAMICBN

dynamic Bayesian Network are a special class of BNs where variables can be subscripted by a (discrete) time.

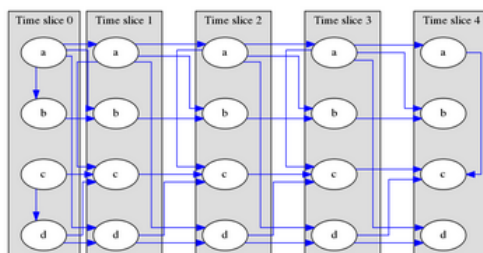
For more details, <<https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/dynamicBn.ipynb.html>>

```
gdyn.showTimeSlices(dbn, format="png")
```



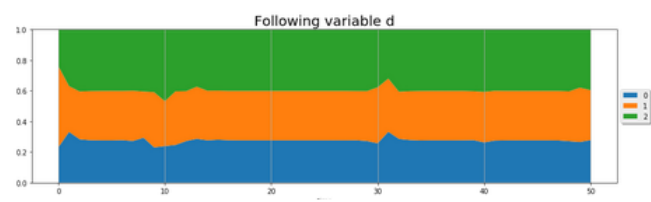
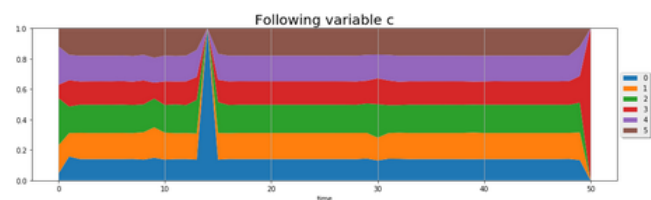
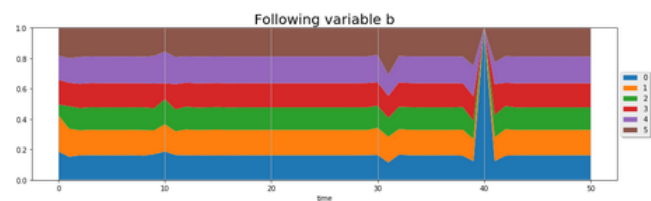
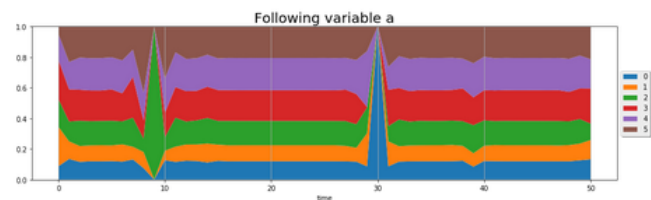
T=5

```
bn=gdyn.unroll2TBN(dbn,T)
gdyn.showTimeSlices(bn,size="10")
```



```
import matplotlib.pyplot as plt
```

```
plt.rcParams['figure.figsize'] = (15, 4)
gdyn.plotFollow(["a", "b", "c", "d"], dbn, T=51, evs={'a9':2, 'a30':0, 'c14':0, 'b40':0, 'c50':3})
```



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** – the dynamic BN
- **size** – size of the fig

`pyAgrum.lib.dynamicBN.getTimeSlicesRange(dbn)`

get the range and (name,radical) of each variables

**Parameters** `dbn` – a 2TBN or an unrolled BN

**Returns** all the timeslice of a dbn

e.g. ['0','t'] for a classic 2TBN range(T) for a classic unrolled BN

`pyAgrum.lib.dynamicBN.is2TBN(bn)`

Check if bn is a 2 TimeSlice Bayesian network

**Parameters** `bn` – the Bayesian network

**Returns** True if the BN is syntactically correct to be a 2TBN

`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** – the dynamic BN
- **size** – size of the fig

`pyAgrum.lib.dynamicBN.unroll2TBN(dbn, nbr)`

unroll a 2TBN given the nbr of timeslices

**Parameters**

- **dbn** – the dBN
- **nbr** – the number of timeslice

**Returns** unrolled BN from a 2TBN and the nbr of timeslices

## OTHER PYAGRUM.LIB MODULES

### 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 58)) – 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 58)) – 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 58)) – 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

## 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`
- **fields** – Dict of name, position in the file
- **csv\_name** – name of the csv file

@throw `gum.DatabaseError` if a BN variable is not in fields

**Returns** return a dictionary of position for BN variables in fields

`pyAgrum.lib.bn2scores.computeScores(bn_name, csv_name, visible=False, transform_label=False)`  
Compute scores from a bn w.r.t to a csv :param `bn_name`: a `gum.BayesianNetwork` or a filename for a BN :param `csv_name`: a filename for the CSV database :param `visible`: do we show the progress :param `transform_label`: do we adapt from labels to id :return: percentDatabaseUsed, scores

`pyAgrum.lib.bn2scores.lines_count(filename)`  
count lines in a file

## 15.3 bn\_vs\_bn

The purpose of this module is to provide tools for comparing 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 58)) – a BN or a filename for reference
- **name2** (*str* or `pyAgrum.BayesNet` (page 58)) – another BN or another filename 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 dictionary 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.

$\text{dist2opt} = \text{square root of } (1 - \text{precision})^2 + (1 - \text{recall})^2$  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.

$\text{dist2opt} = \text{square root of } (1 - \text{precision})^2 + (1 - \text{recall})^2$  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]`





## FUNCTIONS FROM PYAGRUM

### 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 58) or `pyAgrum.MarkovNet` (page 212)) – 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 48) or other)

### 16.2 Quick specification of (randomly parameterized) graphical models

aGrUM/pyAgrum offers a compact syntax that allows to quickly specify prototypes of graphical models. These *fastPrototype* aGrUM's methods have also been wrapped in functions of pyAgrum.

`gum.fastBN("A->B<-C;B->D")`

The type of the random variables can be specify with different syntaxes:

- by default, a variable is a `pyAgrum.RangeVariable` (page 36) using the default domain size (second argument of the functions).
- with a `[10]`, the variable is a `pyAgrum.RangeVariable` (page 36) using 10 as domain size (from 0 to 9)
- with a `[3, 7]`, the variable is a `pyAgrum.RangeVariable` (page 36) using a domainSize from 3 to 7
- with a `[1, 3.14, 5, 6.2]`, the variable is a `pyAgrum.DiscretizedVariable` (page 30) 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 33) 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 58)

`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 212)

`pyAgrum.fastID(structure, domain_size=2)`

**Create an Influence Diagram with a modified dot-like syntax which specifies:**

- the structure 'a->b<-c;b->d;c<-e;',
- the type of the variables with different syntax (cf documentation),
- 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 184)

## 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** – the name of the input file
- **listeners** – list of functions to execute
- **verbose** – whether to print or not warning messages
- **system** – (for O3PRM) name of the system to flatten in a BN
- **classpath** – (for O3PRM) list of folders containing classes

**Returns** a BN from a file using one of the availableBNExts() 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:
```

(continues on next page)

(continued from previous page)

```

>>>     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**(`gum.BayesNet`) – the BN to save
- **filename**(`str`) – the name of the output file

## 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.

`pyAgrum.loadMN(filename, listeners=None, verbose=False)`

load a MN from a file with optional listeners and arguments

#### Parameters

- **filename** – the name of the input file
- **listeners** – list of functions to execute
- **verbose** – whether to print or not warning messages

**Returns** a MN from a file using one of the available `MNExts()` 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)

```

(continues on next page)

(continued from previous page)

```

>>>
>>> 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 available `WriteMNExts()` suffixes.

#### Parameters

- **mn(gum.MarkovNet)** – the MN to save
- **filename(str)** – the name of the output file

## 16.5 Input for influence diagram

**pyAgrum.availableIDExts()**

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.

**pyAgrum.loadID(*filename*)**

read a `gum.InfluenceDiagram` from a ID file

**Parameters** **filename** – the name of the input file

**Returns** an `InfluenceDiagram`

**pyAgrum.saveID(*infdiag*, *filename*)**

save an ID into a file using the format corresponding to one of the available `WriteIDExts()` suffixes.

#### Parameters

- **ID(gum.InfluenceDiagram)** – the ID to save
- **filename(str)** – the name of the output file



## OTHER FUNCTIONS FROM AGRUM

### 17.1 Listeners

aGrUM includes a mechanism for listening to actions (close to QT signal/slot). Some of them have been ported to pyAgrum :

#### 17.1.1 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
```

### 17.1.2 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
```

### 17.1.3 ApproximationSchemeListener

### 17.1.4 DatabaseGenerationListener

## 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.

## 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()`

**Returns** the number of logical processors

**Return type** int

`pyAgrum.getMaxNumberOfThreads()`

**Returns** the max number of threads

**Return type** int



## 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** pyAgrim.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** pyAgrim.UndefinedElement(\*args)

args

**property** thisown

The membership flag

what()

**Return type** str

**exception** pyAgrim.UndefinedIteratorKey(\*args)

args

**property** thisown

The membership flag

what()

**Return type** str

**exception** pyAgrim.UndefinedIteratorValue(\*args)

args

**property** thisown

The membership flag

what()

**Return type** str

**exception** pyAgrim.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



## 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 section name or the property name does not exist

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### p

`pyAgrum.causal.notebook`, [247](#)



## A

- `about()` (in module `pyAgrum`), 277
- `abs()` (`pyAgrum.Potential` method), 48
- `add()` (`pyAgrum.BayesNet` method), 58
- `add()` (`pyAgrum.InfluenceDiagram` method), 184
- `add()` (`pyAgrum.Instantiation` method), 42
- `add()` (`pyAgrum.MarkovNet` method), 212
- `add()` (`pyAgrum.Potential` method), 48
- `add_hook()` (`pyAgrum.PyAgrumConfiguration` method), 293
- `addAllTargets()` (`pyAgrum.GibbsSampling` method), 117
- `addAllTargets()` (`pyAgrum.ImportanceSampling` method), 138
- `addAllTargets()` (`pyAgrum.LazyPropagation` method), 91
- `addAllTargets()` (`pyAgrum.LoopyBeliefPropagation` method), 111
- `addAllTargets()` (`pyAgrum.LoopyGibbsSampling` method), 145
- `addAllTargets()` (`pyAgrum.LoopyImportanceSampling` method), 166
- `addAllTargets()` (`pyAgrum.LoopyMonteCarloSampling` method), 152
- `addAllTargets()` (`pyAgrum.LoopyWeightedSampling` method), 159
- `addAllTargets()` (`pyAgrum.MonteCarloSampling` method), 124
- `addAllTargets()` (`pyAgrum.ShaferShenoyInference` method), 98
- `addAllTargets()` (`pyAgrum.ShaferShenoyMNIInference` method), 218
- `addAllTargets()` (`pyAgrum.VariableElimination` method), 104
- `addAllTargets()` (`pyAgrum.WeightedSampling` method), 131
- `addAMPLITUDE()` (`pyAgrum.BayesNet` method), 58
- `addAND()` (`pyAgrum.BayesNet` method), 58
- `addArc()` (`pyAgrum.BayesNet` method), 58
- `addArc()` (`pyAgrum.CredalNet` method), 197
- `addArc()` (`pyAgrum.DAG` method), 8
- `addArc()` (`pyAgrum.DiGraph` method), 4
- `addArc()` (`pyAgrum.InfluenceDiagram` method), 184
- `addArc()` (`pyAgrum.MixedGraph` method), 18
- `addCausalArc()` (`pyAgrum.causal.CausalModel` method), 234
- `addChanceNode()` (`pyAgrum.InfluenceDiagram` method), 184
- `addCOUNT()` (`pyAgrum.BayesNet` method), 59
- `addDecisionNode()` (`pyAgrum.InfluenceDiagram` method), 185
- `addEdge()` (`pyAgrum.CliqueGraph` method), 14
- `addEdge()` (`pyAgrum.MixedGraph` method), 19
- `addEdge()` (`pyAgrum.UndiGraph` method), 11
- `addEvidence()` (`pyAgrum.GibbsSampling` method), 117
- `addEvidence()` (`pyAgrum.ImportanceSampling` method), 138
- `addEvidence()` (`pyAgrum.LazyPropagation` method), 91
- `addEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 111
- `addEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 145
- `addEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 166
- `addEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 152
- `addEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 159
- `addEvidence()` (`pyAgrum.MonteCarloSampling` method), 125
- `addEvidence()` (`pyAgrum.ShaferShenoyInference` method), 98
- `addEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 192
- `addEvidence()` (`pyAgrum.ShaferShenoyMNIInference` method), 218
- `addEvidence()` (`pyAgrum.VariableElimination` method), 104
- `addEvidence()` (`pyAgrum.WeightedSampling` method), 131
- `addEXISTS()` (`pyAgrum.BayesNet` method), 59
- `addFactor()` (`pyAgrum.MarkovNet` method), 212
- `addFORALL()` (`pyAgrum.BayesNet` method), 59
- `addForbiddenArc()` (`pyAgrum.BN Learner` method),

- 173
- `addJointTarget()` (*pyAgrum.LazyPropagation method*), 91
- `addJointTarget()` (*pyAgrum.ShaferShenoyInference method*), 98
- `addJointTarget()` (*pyAgrum.ShaferShenoyMNIInference method*), 219
- `addJointTarget()` (*pyAgrum.VariableElimination method*), 105
- `addLabel()` (*pyAgrum.LabelizedVariable method*), 28
- `addLatentVariable()` (*pyAgrum.causal.CausalModel method*), 234
- `addLogit()` (*pyAgrum.BayesNet method*), 59
- `addMandatoryArc()` (*pyAgrum.BN Learner method*), 173
- `addMAX()` (*pyAgrum.BayesNet method*), 60
- `addMEDIAN()` (*pyAgrum.BayesNet method*), 60
- `addMIN()` (*pyAgrum.BayesNet method*), 60
- `addNode()` (*pyAgrum.CliqueGraph method*), 14
- `addNode()` (*pyAgrum.DAG method*), 8
- `addNode()` (*pyAgrum.DiGraph method*), 5
- `addNode()` (*pyAgrum.MixedGraph method*), 19
- `addNode()` (*pyAgrum.UndiGraph method*), 11
- `addNodes()` (*pyAgrum.CliqueGraph method*), 14
- `addNodes()` (*pyAgrum.DAG method*), 8
- `addNodes()` (*pyAgrum.DiGraph method*), 5
- `addNodes()` (*pyAgrum.MixedGraph method*), 19
- `addNodes()` (*pyAgrum.UndiGraph method*), 11
- `addNodeWithId()` (*pyAgrum.CliqueGraph method*), 14
- `addNodeWithId()` (*pyAgrum.DAG method*), 8
- `addNodeWithId()` (*pyAgrum.DiGraph method*), 5
- `addNodeWithId()` (*pyAgrum.MixedGraph method*), 19
- `addNodeWithId()` (*pyAgrum.UndiGraph method*), 11
- `addNoForgettingAssumption()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 192
- `addNoisyAND()` (*pyAgrum.BayesNet method*), 60
- `addNoisyOR()` (*pyAgrum.BayesNet method*), 60
- `addNoisyORCompound()` (*pyAgrum.BayesNet method*), 61
- `addNoisyORNet()` (*pyAgrum.BayesNet method*), 61
- `addOR()` (*pyAgrum.BayesNet method*), 61
- `addPossibleEdge()` (*pyAgrum.BN Learner method*), 174
- `addStructureListener()` (*pyAgrum.BayesNet method*), 62
- `addStructureListener()` (*pyAgrum.BayesNetFragment method*), 82
- `addStructureListener()` (*pyAgrum.MarkovNet method*), 212
- `addSUM()` (*pyAgrum.BayesNet method*), 62
- `addTarget()` (*pyAgrum.GibbsSampling method*), 118
- `addTarget()` (*pyAgrum.ImportanceSampling method*), 138
- `addTarget()` (*pyAgrum.LazyPropagation method*), 91
- `addTarget()` (*pyAgrum.LoopyBeliefPropagation method*), 111
- `addTarget()` (*pyAgrum.LoopyGibbsSampling method*), 145
- `addTarget()` (*pyAgrum.LoopyImportanceSampling method*), 166
- `addTarget()` (*pyAgrum.LoopyMonteCarloSampling method*), 153
- `addTarget()` (*pyAgrum.LoopyWeightedSampling method*), 159
- `addTarget()` (*pyAgrum.MonteCarloSampling method*), 125
- `addTarget()` (*pyAgrum.ShaferShenoyInference method*), 98
- `addTarget()` (*pyAgrum.ShaferShenoyMNIInference method*), 219
- `addTarget()` (*pyAgrum.VariableElimination method*), 105
- `addTarget()` (*pyAgrum.WeightedSampling method*), 132
- `addTick()` (*pyAgrum.DiscretizedVariable method*), 31
- `addToClique()` (*pyAgrum.CliqueGraph method*), 14
- `addUtilityNode()` (*pyAgrum.InfluenceDiagram method*), 185
- `addValue()` (*pyAgrum.IntegerVariable method*), 33
- `addVariable()` (*pyAgrum.CredalNet method*), 198
- `addVarsFromModel()` (*pyAgrum.Instantiation method*), 42
- `addWeightedArc()` (*pyAgrum.BayesNet method*), 62
- `adjacents()` (*pyAgrum.MixedGraph method*), 19
- `aggType` (*pyAgrum.PRMexplorer property*), 227
- `ancestors()` (*pyAgrum.BayesNet method*), 62
- `ancestors()` (*pyAgrum.BayesNetFragment method*), 83
- `ancestors()` (*pyAgrum.InfluenceDiagram method*), 185
- `animApproximationScheme()` (in module *pyAgrum.lib.notebook*), 264
- `approximatedBinarization()` (*pyAgrum.CredalNet method*), 198
- `Arc` (class in *pyAgrum*), 3
- `arcs()` (*pyAgrum.BayesNet method*), 62
- `arcs()` (*pyAgrum.BayesNetFragment method*), 83
- `arcs()` (*pyAgrum.causal.CausalModel method*), 234
- `arcs()` (*pyAgrum.DAG method*), 8
- `arcs()` (*pyAgrum.DiGraph method*), 5
- `arcs()` (*pyAgrum.EssentialGraph method*), 79
- `arcs()` (*pyAgrum.InfluenceDiagram method*), 185
- `arcs()` (*pyAgrum.MarkovBlanket method*), 81
- `arcs()` (*pyAgrum.MixedGraph method*), 19
- `argmax()` (*pyAgrum.Potential method*), 48
- `argmin()` (*pyAgrum.Potential method*), 48
- `args` (*pyAgrum.ArgumentError attribute*), 290
- `args` (*pyAgrum.causal.HedgeException attribute*), 246
- `args` (*pyAgrum.causal.UnidentifiableException attribute*), 247
- `args` (*pyAgrum.CPTErrror 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), 288  
 args (*pyAgrum.FormatNotFound* attribute), 288  
 args (*pyAgrum.GraphError* attribute), 288  
 args (*pyAgrum.GumException* attribute), 287  
 args (*pyAgrum.InvalidArc* attribute), 288  
 args (*pyAgrum.InvalidArgument* attribute), 288  
 args (*pyAgrum.InvalidArgumentsNumber* attribute), 289  
 args (*pyAgrum.InvalidDirectedCycle* attribute), 289  
 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), 290  
 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*), 239  
 ASTdiv (class in *pyAgrum.causal*), 242  
 ASTjointProba (class in *pyAgrum.causal*), 245  
 ASTminus (class in *pyAgrum.causal*), 241  
 ASTmult (class in *pyAgrum.causal*), 243  
 ASTplus (class in *pyAgrum.causal*), 240  
 ASTposteriorProba (class in *pyAgrum.causal*), 245  
 ASTsum (class in *pyAgrum.causal*), 244  
 ASTtree (class in *pyAgrum.causal*), 239  
 audit() (*pyAgrum.skbn.BNDiscretizer* method), 253  
 availableBNExts() (in module *pyAgrum*), 279  
 availableIDExtS() (in module *pyAgrum*), 281  
 availableMNExts() (in module *pyAgrum*), 280

## B

backDoor() (*pyAgrum.causal.CausalModel* method), 234  
 BayesNet (class in *pyAgrum*), 58  
 BayesNetFragment (class in *pyAgrum*), 82  
 beginTopologyTransformation() (*pyAgrum.BayesNet* method), 62  
 beginTopologyTransformation() (*pyAgrum.MarkovNet* method), 212  
 belongs() (*pyAgrum.RangeVariable* method), 36  
 binaryJoinTree() (*pyAgrum.JunctionTreeGenerator* method), 78  
 bn (*pyAgrum.causal.ASTposteriorProba* property), 245  
 BN() (*pyAgrum.GibbsSampling* method), 117

BN() (*pyAgrum.ImportanceSampling* method), 138  
 BN() (*pyAgrum.LazyPropagation* method), 90  
 BN() (*pyAgrum.LoopyBeliefPropagation* method), 110  
 BN() (*pyAgrum.LoopyGibbsSampling* method), 144  
 BN() (*pyAgrum.LoopyImportanceSampling* method), 166  
 BN() (*pyAgrum.LoopyMonteCarloSampling* method), 152  
 BN() (*pyAgrum.LoopyWeightedSampling* method), 159  
 BN() (*pyAgrum.MonteCarloSampling* method), 124  
 BN() (*pyAgrum.ShaferShenoyInference* method), 97  
 BN() (*pyAgrum.VariableElimination* method), 104  
 BN() (*pyAgrum.WeightedSampling* method), 131  
 BNClassifier (class in *pyAgrum.skbn*), 250  
 BNDatabaseGenerator (class in *pyAgrum*), 72  
 BNDiscretizer (class in *pyAgrum.skbn*), 253  
 BNLearner (class in *pyAgrum*), 173  
 bnToCredal() (*pyAgrum.CredalNet* method), 198  
 burnIn() (*pyAgrum.GibbsBNDistance* method), 74  
 burnIn() (*pyAgrum.GibbsSampling* method), 118  
 burnIn() (*pyAgrum.LoopyGibbsSampling* method), 145

## C

causalBN() (*pyAgrum.causal.CausalModel* method), 234  
 CausalFormula (class in *pyAgrum.causal*), 236  
 causalImpact() (in module *pyAgrum.causal*), 237  
 CausalModel (class in *pyAgrum.causal*), 234  
 chanceNodeSize() (*pyAgrum.InfluenceDiagram* method), 185  
 changeLabel() (*pyAgrum.LabelizedVariable* method), 28  
 changePotential() (*pyAgrum.BayesNet* method), 62  
 changeValue() (*pyAgrum.IntegerVariable* method), 33  
 changeVariableLabel() (*pyAgrum.BayesNet* method), 63  
 changeVariableLabel() (*pyAgrum.MarkovNet* method), 213  
 changeVariableName() (*pyAgrum.BayesNet* method), 63  
 changeVariableName() (*pyAgrum.InfluenceDiagram* method), 186  
 changeVariableName() (*pyAgrum.MarkovNet* method), 213  
 checkConsistency() (*pyAgrum.BayesNetFragment* method), 83  
 chgEvidence() (*pyAgrum.GibbsSampling* method), 118  
 chgEvidence() (*pyAgrum.ImportanceSampling* method), 139  
 chgEvidence() (*pyAgrum.LazyPropagation* method), 92  
 chgEvidence() (*pyAgrum.LoopyBeliefPropagation* method), 111  
 chgEvidence() (*pyAgrum.LoopyGibbsSampling* method), 146

- chgEvidence() (*pyAgrum.LoopyImportanceSampling method*), 167
- chgEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 153
- chgEvidence() (*pyAgrum.LoopyWeightedSampling method*), 160
- chgEvidence() (*pyAgrum.MonteCarloSampling method*), 125
- chgEvidence() (*pyAgrum.ShaferShenoyInference method*), 99
- chgEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 192
- chgEvidence() (*pyAgrum.ShaferShenoyMNIInference method*), 219
- chgEvidence() (*pyAgrum.VariableElimination method*), 105
- chgEvidence() (*pyAgrum.WeightedSampling method*), 132
- chgVal() (*pyAgrum.Instantiation method*), 43
- chi2() (*pyAgrum.BN Learner method*), 174
- children() (*pyAgrum.BayesNet method*), 63
- children() (*pyAgrum.BayesNetFragment method*), 83
- children() (*pyAgrum.causal.CausalModel method*), 235
- children() (*pyAgrum.DAG method*), 8
- children() (*pyAgrum.DiGraph method*), 5
- children() (*pyAgrum.EssentialGraph method*), 79
- children() (*pyAgrum.InfluenceDiagram method*), 186
- children() (*pyAgrum.MarkovBlanket method*), 81
- children() (*pyAgrum.MixedGraph method*), 19
- classAggregates() (*pyAgrum.PRMexplorer method*), 227
- classAttributes() (*pyAgrum.PRMexplorer method*), 227
- classDag() (*pyAgrum.PRMexplorer method*), 227
- classes() (*pyAgrum.PRMexplorer method*), 228
- classImplements() (*pyAgrum.PRMexplorer method*), 227
- classParameters() (*pyAgrum.PRMexplorer method*), 227
- classReferences() (*pyAgrum.PRMexplorer method*), 228
- classSlotChains() (*pyAgrum.PRMexplorer method*), 228
- clear() (*pyAgrum.BayesNet method*), 63
- clear() (*pyAgrum.CliqueGraph method*), 15
- clear() (*pyAgrum.DAG method*), 8
- clear() (*pyAgrum.DiGraph method*), 5
- clear() (*pyAgrum.InfluenceDiagram method*), 186
- clear() (*pyAgrum.Instantiation method*), 43
- clear() (*pyAgrum.MarkovNet method*), 213
- clear() (*pyAgrum.MixedGraph method*), 19
- clear() (*pyAgrum.ShaferShenoyLIMIDInference method*), 192
- clear() (*pyAgrum.skbn.BNDiscretizer method*), 254
- clear() (*pyAgrum.UndiGraph method*), 11
- clearEdges() (*pyAgrum.CliqueGraph method*), 15
- clique() (*pyAgrum.CliqueGraph method*), 15
- CliqueGraph (class in *pyAgrum*), 14
- cm (*pyAgrum.causal.CausalFormula property*), 236
- CN() (*pyAgrum.CNLoopyPropagation method*), 205
- CN() (*pyAgrum.CNMonteCarloSampling method*), 202
- CNLoopyPropagation (class in *pyAgrum*), 205
- CNMonteCarloSampling (class in *pyAgrum*), 202
- col() (*pyAgrum.SyntaxError method*), 291
- completeInstantiation() (*pyAgrum.BayesNet method*), 63
- completeInstantiation() (*pyAgrum.BayesNetFragment method*), 83
- completeInstantiation() (*pyAgrum.InfluenceDiagram method*), 186
- completeInstantiation() (*pyAgrum.MarkovNet method*), 213
- compute() (*pyAgrum.ExactBNdistance method*), 74
- compute() (*pyAgrum.GibbsBNdistance method*), 74
- computeBinaryCPTMinMax() (*pyAgrum.CredalNet method*), 198
- configuration() (in module *pyAgrum.lib.notebook*), 264
- connectedComponents() (*pyAgrum.BayesNet method*), 64
- connectedComponents() (*pyAgrum.BayesNetFragment method*), 83
- connectedComponents() (*pyAgrum.CliqueGraph method*), 15
- connectedComponents() (*pyAgrum.DAG method*), 9
- connectedComponents() (*pyAgrum.DiGraph method*), 5
- connectedComponents() (*pyAgrum.EssentialGraph method*), 79
- connectedComponents() (*pyAgrum.InfluenceDiagram method*), 186
- connectedComponents() (*pyAgrum.MarkovBlanket method*), 81
- connectedComponents() (*pyAgrum.MarkovNet method*), 213
- connectedComponents() (*pyAgrum.MixedGraph method*), 20
- connectedComponents() (*pyAgrum.UndiGraph method*), 12
- container() (*pyAgrum.CliqueGraph method*), 15
- containerPath() (*pyAgrum.CliqueGraph method*), 15
- contains() (*pyAgrum.Instantiation method*), 43
- contains() (*pyAgrum.Potential method*), 48
- continueApproximationScheme() (*pyAgrum.GibbsBNdistance method*), 74
- copy() (*pyAgrum.causal.ASTBinaryOp method*), 239
- copy() (*pyAgrum.causal.ASTdiv method*), 242
- copy() (*pyAgrum.causal.ASTjointProba method*), 245
- copy() (*pyAgrum.causal.ASTminus method*), 241
- copy() (*pyAgrum.causal.ASTmult method*), 243
- copy() (*pyAgrum.causal.ASTplus method*), 240

- copy() (*pyAgrum.causal.ASTposteriorProba* method), 246
- copy() (*pyAgrum.causal.ASTsum* method), 244
- copy() (*pyAgrum.causal.ASTtree* method), 239
- copy() (*pyAgrum.causal.CausalFormula* method), 236
- cpf() (*pyAgrum.PRMexplorer* method), 228
- cpt() (*pyAgrum.BayesNet* method), 64
- cpt() (*pyAgrum.BayesNetFragment* method), 83
- cpt() (*pyAgrum.InfluenceDiagram* method), 186
- CPTError, 292
- createVariable() (*pyAgrum.skbn.BNDiscretizer* method), 254
- CredalNet (class in *pyAgrum*), 197
- credalNet\_currentCpt() (*pyAgrum.CredalNet* method), 198
- credalNet\_srcCpt() (*pyAgrum.CredalNet* method), 198
- current\_bn() (*pyAgrum.CredalNet* method), 199
- currentNodeType() (*pyAgrum.CredalNet* method), 198
- currentPosterior() (*pyAgrum.GibbsSampling* method), 118
- currentPosterior() (*pyAgrum.ImportanceSampling* method), 139
- currentPosterior() (*pyAgrum.LoopyGibbsSampling* method), 146
- currentPosterior() (*pyAgrum.LoopyImportanceSampling* method), 167
- currentPosterior() (*pyAgrum.LoopyMonteCarloSampling* method), 153
- currentPosterior() (*pyAgrum.LoopyWeightedSampling* method), 160
- currentPosterior() (*pyAgrum.MonteCarloSampling* method), 126
- currentPosterior() (*pyAgrum.WeightedSampling* method), 132
- currentTime() (*pyAgrum.BNlearner* method), 174
- currentTime() (*pyAgrum.CNLoopyPropagation* method), 206
- currentTime() (*pyAgrum.CNMonteCarloSampling* method), 203
- currentTime() (*pyAgrum.GibbsBNdistance* method), 74
- currentTime() (*pyAgrum.GibbsSampling* method), 119
- currentTime() (*pyAgrum.ImportanceSampling* method), 139
- currentTime() (*pyAgrum.LoopyBeliefPropagation* method), 112
- currentTime() (*pyAgrum.LoopyGibbsSampling* method), 146
- currentTime() (*pyAgrum.LoopyImportanceSampling* method), 167
- currentTime() (*pyAgrum.LoopyMonteCarloSampling* method), 153
- currentTime() (*pyAgrum.LoopyWeightedSampling* method), 160
- currentTime() (*pyAgrum.MonteCarloSampling* method), 126
- currentTime() (*pyAgrum.WeightedSampling* method), 132
- ## D
- DAG (class in *pyAgrum*), 8
- dag() (*pyAgrum.BayesNet* method), 64
- dag() (*pyAgrum.BayesNetFragment* method), 84
- dag() (*pyAgrum.InfluenceDiagram* method), 186
- dag() (*pyAgrum.MarkovBlanket* method), 81
- database() (*pyAgrum.BNDatabaseGenerator* method), 72
- DatabaseError, 292
- databaseWeight() (*pyAgrum.BNlearner* method), 174
- dec() (*pyAgrum.Instantiation* method), 43
- decIn() (*pyAgrum.Instantiation* method), 43
- decisionNodeSize() (*pyAgrum.InfluenceDiagram* method), 187
- decisionOrder() (*pyAgrum.InfluenceDiagram* method), 187
- decisionOrderExists() (*pyAgrum.InfluenceDiagram* method), 187
- decNotVar() (*pyAgrum.Instantiation* method), 43
- decOut() (*pyAgrum.Instantiation* method), 43
- decVar() (*pyAgrum.Instantiation* method), 43
- DefaultInLabel, 287
- descendants() (*pyAgrum.BayesNet* method), 64
- descendants() (*pyAgrum.BayesNetFragment* method), 84
- descendants() (*pyAgrum.InfluenceDiagram* method), 187
- description() (*pyAgrum.DiscreteVariable* method), 25
- description() (*pyAgrum.DiscretizedVariable* method), 31
- description() (*pyAgrum.IntegerVariable* method), 34
- description() (*pyAgrum.LabelizedVariable* method), 28
- description() (*pyAgrum.RangeVariable* method), 37
- diff() (*pyAgrum.PyAgrumConfiguration* method), 293
- DiGraph (class in *pyAgrum*), 4
- dim() (*pyAgrum.BayesNet* method), 64
- dim() (*pyAgrum.BayesNetFragment* method), 84
- dim() (*pyAgrum.MarkovNet* method), 213
- disableEpsilon() (*pyAgrum.GibbsBNdistance* method), 74
- disableMaxIter() (*pyAgrum.GibbsBNdistance* method), 74
- disableMaxTime() (*pyAgrum.GibbsBNdistance* method), 74



- [disableMinEpsilonRate\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[DiscreteVariable](#) (class in *pyAgrum*), 25  
[discretizationCAIM\(\)](#) (*pyAgrum.skbn.BNDiscretizer method*), 254  
[discretizationElbowMethodRotation\(\)](#) (*pyAgrum.skbn.BNDiscretizer method*), 254  
[discretizationMDLP\(\)](#) (*pyAgrum.skbn.BNDiscretizer method*), 255  
[discretizationNML\(\)](#) (*pyAgrum.skbn.BNDiscretizer method*), 255  
[DiscretizedVariable](#) (class in *pyAgrum*), 30  
[doCalculusWithObservation\(\)](#) (in module *pyAgrum.causal*), 237  
[domain\(\)](#) (*pyAgrum.DiscreteVariable method*), 25  
[domain\(\)](#) (*pyAgrum.DiscretizedVariable method*), 31  
[domain\(\)](#) (*pyAgrum.IntegerVariable method*), 34  
[domain\(\)](#) (*pyAgrum.LabelizedVariable method*), 28  
[domain\(\)](#) (*pyAgrum.RangeVariable method*), 37  
[domainSize\(\)](#) (*pyAgrum.BN Learner method*), 174  
[domainSize\(\)](#) (*pyAgrum.CredalNet method*), 199  
[domainSize\(\)](#) (*pyAgrum.DiscreteVariable method*), 25  
[domainSize\(\)](#) (*pyAgrum.DiscretizedVariable method*), 31  
[domainSize\(\)](#) (*pyAgrum.Instantiation method*), 44  
[domainSize\(\)](#) (*pyAgrum.IntegerVariable method*), 34  
[domainSize\(\)](#) (*pyAgrum.LabelizedVariable method*), 28  
[domainSize\(\)](#) (*pyAgrum.Potential method*), 48  
[domainSize\(\)](#) (*pyAgrum.RangeVariable method*), 37  
[draw\(\)](#) (*pyAgrum.Potential method*), 49  
[drawSamples\(\)](#) (*pyAgrum.BNDatabaseGenerator method*), 72  
[dSeparation\(\)](#) (*pyAgrum.DAG method*), 9  
[DuplicateElement](#), 287  
[DuplicateLabel](#), 287  
[dynamicExpMax\(\)](#) (*pyAgrum.CNLoopyPropagation method*), 206  
[dynamicExpMax\(\)](#) (*pyAgrum.CNMonteCarloSampling method*), 203  
[dynamicExpMin\(\)](#) (*pyAgrum.CNLoopyPropagation method*), 206  
[dynamicExpMin\(\)](#) (*pyAgrum.CNMonteCarloSampling method*), 203
- ## E
- [Edge](#) (class in *pyAgrum*), 4  
[edges\(\)](#) (*pyAgrum.CliqueGraph method*), 15  
[edges\(\)](#) (*pyAgrum.EssentialGraph method*), 80  
[edges\(\)](#) (*pyAgrum.MarkovNet method*), 213  
[edges\(\)](#) (*pyAgrum.MixedGraph method*), 20  
[edges\(\)](#) (*pyAgrum.UndiGraph method*), 12  
[eliminationOrder\(\)](#) (*pyAgrum.JunctionTreeGenerator method*), 79  
[empty\(\)](#) (*pyAgrum.BayesNet method*), 64  
[empty\(\)](#) (*pyAgrum.BayesNetFragment method*), 84  
[empty\(\)](#) (*pyAgrum.CliqueGraph method*), 15  
[empty\(\)](#) (*pyAgrum.DAG method*), 9  
[empty\(\)](#) (*pyAgrum.DiGraph method*), 6  
[empty\(\)](#) (*pyAgrum.DiscreteVariable method*), 25  
[empty\(\)](#) (*pyAgrum.DiscretizedVariable method*), 31  
[empty\(\)](#) (*pyAgrum.InfluenceDiagram method*), 187  
[empty\(\)](#) (*pyAgrum.Instantiation method*), 44  
[empty\(\)](#) (*pyAgrum.IntegerVariable method*), 34  
[empty\(\)](#) (*pyAgrum.LabelizedVariable method*), 28  
[empty\(\)](#) (*pyAgrum.MarkovNet method*), 214  
[empty\(\)](#) (*pyAgrum.MixedGraph method*), 20  
[empty\(\)](#) (*pyAgrum.Potential method*), 49  
[empty\(\)](#) (*pyAgrum.RangeVariable method*), 37  
[empty\(\)](#) (*pyAgrum.UndiGraph method*), 12  
[emptyArcs\(\)](#) (*pyAgrum.DAG method*), 9  
[emptyArcs\(\)](#) (*pyAgrum.DiGraph method*), 6  
[emptyArcs\(\)](#) (*pyAgrum.MixedGraph method*), 20  
[emptyEdges\(\)](#) (*pyAgrum.CliqueGraph method*), 16  
[emptyEdges\(\)](#) (*pyAgrum.MixedGraph method*), 20  
[emptyEdges\(\)](#) (*pyAgrum.UndiGraph method*), 12  
[enableEpsilon\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[enableMaxIter\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[enableMaxTime\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[enableMinEpsilonRate\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[end\(\)](#) (*pyAgrum.Instantiation method*), 44  
[endTopologyTransformation\(\)](#) (*pyAgrum.BayesNet method*), 64  
[endTopologyTransformation\(\)](#) (*pyAgrum.MarkovNet method*), 214  
[entropy\(\)](#) (*pyAgrum.Potential method*), 49  
[epsilon\(\)](#) (*pyAgrum.BN Learner method*), 174  
[epsilon\(\)](#) (*pyAgrum.CNLoopyPropagation method*), 206  
[epsilon\(\)](#) (*pyAgrum.CNMonteCarloSampling method*), 203  
[epsilon\(\)](#) (*pyAgrum.GibbsBNdistance method*), 75  
[epsilon\(\)](#) (*pyAgrum.GibbsSampling method*), 119  
[epsilon\(\)](#) (*pyAgrum.ImportanceSampling method*), 139  
[epsilon\(\)](#) (*pyAgrum.LoopyBeliefPropagation method*), 112  
[epsilon\(\)](#) (*pyAgrum.LoopyGibbsSampling method*), 146  
[epsilon\(\)](#) (*pyAgrum.LoopyImportanceSampling method*), 167  
[epsilon\(\)](#) (*pyAgrum.LoopyMonteCarloSampling method*), 154  
[epsilon\(\)](#) (*pyAgrum.LoopyWeightedSampling method*), 160  
[epsilon\(\)](#) (*pyAgrum.MonteCarloSampling method*), 126  
[epsilon\(\)](#) (*pyAgrum.WeightedSampling method*), 133

- [epsilonMax\(\)](#) ([pyAgrum.CredalNet method](#)), 199  
[epsilonMean\(\)](#) ([pyAgrum.CredalNet method](#)), 199  
[epsilonMin\(\)](#) ([pyAgrum.CredalNet method](#)), 199  
[erase\(\)](#) ([pyAgrum.BayesNet method](#)), 64  
[erase\(\)](#) ([pyAgrum.InfluenceDiagram method](#)), 187  
[erase\(\)](#) ([pyAgrum.Instantiation method](#)), 44  
[erase\(\)](#) ([pyAgrum.MarkovNet method](#)), 214  
[eraseAllEvidence\(\)](#) ([pyAgrum.CNLoopyPropagation method](#)), 206  
[eraseAllEvidence\(\)](#) ([pyAgrum.GibbsSampling method](#)), 119  
[eraseAllEvidence\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 139  
[eraseAllEvidence\(\)](#) ([pyAgrum.LazyPropagation method](#)), 92  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 112  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 146  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyImportanceSampling method](#)), 167  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyMonteCarloSampling method](#)), 154  
[eraseAllEvidence\(\)](#) ([pyAgrum.LoopyWeightedSampling method](#)), 160  
[eraseAllEvidence\(\)](#) ([pyAgrum.MonteCarloSampling method](#)), 126  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 99  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyLIMIDInference method](#)), 192  
[eraseAllEvidence\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 220  
[eraseAllEvidence\(\)](#) ([pyAgrum.VariableElimination method](#)), 106  
[eraseAllEvidence\(\)](#) ([pyAgrum.WeightedSampling method](#)), 133  
[eraseAllJointTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 92  
[eraseAllJointTargets\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 99  
[eraseAllJointTargets\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 220  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 92  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 99  
[eraseAllMarginalTargets\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 220  
[eraseAllTargets\(\)](#) ([pyAgrum.GibbsSampling method](#)), 119  
[eraseAllTargets\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 140  
[eraseAllTargets\(\)](#) ([pyAgrum.LazyPropagation method](#)), 92  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 112  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 146  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyImportanceSampling method](#)), 168  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyMonteCarloSampling method](#)), 154  
[eraseAllTargets\(\)](#) ([pyAgrum.LoopyWeightedSampling method](#)), 161  
[eraseAllTargets\(\)](#) ([pyAgrum.MonteCarloSampling method](#)), 126  
[eraseAllTargets\(\)](#) ([pyAgrum.ShaferShenoyInference method](#)), 99  
[eraseAllTargets\(\)](#) ([pyAgrum.ShaferShenoyMNIInference method](#)), 220  
[eraseAllTargets\(\)](#) ([pyAgrum.VariableElimination method](#)), 106  
[eraseAllTargets\(\)](#) ([pyAgrum.WeightedSampling method](#)), 133  
[eraseArc\(\)](#) ([pyAgrum.BayesNet method](#)), 65  
[eraseArc\(\)](#) ([pyAgrum.DAG method](#)), 9  
[eraseArc\(\)](#) ([pyAgrum.DiGraph method](#)), 6  
[eraseArc\(\)](#) ([pyAgrum.InfluenceDiagram method](#)), 187  
[eraseArc\(\)](#) ([pyAgrum.MixedGraph method](#)), 20  
[eraseCausalArc\(\)](#) ([pyAgrum.causal.CausalModel method](#)), 235  
[eraseChildren\(\)](#) ([pyAgrum.DAG method](#)), 9  
[eraseChildren\(\)](#) ([pyAgrum.DiGraph method](#)), 6  
[eraseChildren\(\)](#) ([pyAgrum.MixedGraph method](#)), 20  
[eraseEdge\(\)](#) ([pyAgrum.CliqueGraph method](#)), 16  
[eraseEdge\(\)](#) ([pyAgrum.MixedGraph method](#)), 20  
[eraseEdge\(\)](#) ([pyAgrum.UndiGraph method](#)), 12  
[eraseEvidence\(\)](#) ([pyAgrum.GibbsSampling method](#)), 119  
[eraseEvidence\(\)](#) ([pyAgrum.ImportanceSampling method](#)), 140  
[eraseEvidence\(\)](#) ([pyAgrum.LazyPropagation method](#)), 92  
[eraseEvidence\(\)](#) ([pyAgrum.LoopyBeliefPropagation method](#)), 112  
[eraseEvidence\(\)](#) ([pyAgrum.LoopyGibbsSampling method](#)), 146

- method*), 146
- `eraseEvidence()` (`pyAgrum.LoopyImportanceSampling` *method*), 168
- `eraseEvidence()` (`pyAgrum.LoopyMonteCarloSampling` *method*), 154
- `eraseEvidence()` (`pyAgrum.LoopyWeightedSampling` *method*), 161
- `eraseEvidence()` (`pyAgrum.MonteCarloSampling` *method*), 126
- `eraseEvidence()` (`pyAgrum.ShaferShenoyInference` *method*), 99
- `eraseEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` *method*), 193
- `eraseEvidence()` (`pyAgrum.ShaferShenoyMNIInference` *method*), 220
- `eraseEvidence()` (`pyAgrum.VariableElimination` *method*), 106
- `eraseEvidence()` (`pyAgrum.WeightedSampling` *method*), 133
- `eraseFactor()` (`pyAgrum.MarkovNet` *method*), 214
- `eraseForbiddenArc()` (`pyAgrum.BN Learner` *method*), 174
- `eraseFromClique()` (`pyAgrum.CliqueGraph` *method*), 16
- `eraseJointTarget()` (`pyAgrum.LazyPropagation` *method*), 92
- `eraseJointTarget()` (`pyAgrum.ShaferShenoyInference` *method*), 100
- `eraseJointTarget()` (`pyAgrum.ShaferShenoyMNIInference` *method*), 220
- `eraseJointTarget()` (`pyAgrum.VariableElimination` *method*), 106
- `eraseLabels()` (`pyAgrum.LabelizedVariable` *method*), 28
- `eraseMandatoryArc()` (`pyAgrum.BN Learner` *method*), 174
- `eraseNeighbours()` (`pyAgrum.CliqueGraph` *method*), 16
- `eraseNeighbours()` (`pyAgrum.MixedGraph` *method*), 21
- `eraseNeighbours()` (`pyAgrum.UndiGraph` *method*), 12
- `eraseNode()` (`pyAgrum.CliqueGraph` *method*), 16
- `eraseNode()` (`pyAgrum.DAG` *method*), 9
- `eraseNode()` (`pyAgrum.DiGraph` *method*), 6
- `eraseNode()` (`pyAgrum.MixedGraph` *method*), 21
- `eraseNode()` (`pyAgrum.UndiGraph` *method*), 12
- `eraseParents()` (`pyAgrum.DAG` *method*), 9
- `eraseParents()` (`pyAgrum.DiGraph` *method*), 6
- `eraseParents()` (`pyAgrum.MixedGraph` *method*), 21
- `erasePossibleEdge()` (`pyAgrum.BN Learner` *method*), 175
- `eraseTarget()` (`pyAgrum.GibbsSampling` *method*), 119
- `eraseTarget()` (`pyAgrum.ImportanceSampling` *method*), 140
- `eraseTarget()` (`pyAgrum.LazyPropagation` *method*), 93
- `eraseTarget()` (`pyAgrum.LoopyBeliefPropagation` *method*), 112
- `eraseTarget()` (`pyAgrum.LoopyGibbsSampling` *method*), 147
- `eraseTarget()` (`pyAgrum.LoopyImportanceSampling` *method*), 168
- `eraseTarget()` (`pyAgrum.LoopyMonteCarloSampling` *method*), 154
- `eraseTarget()` (`pyAgrum.LoopyWeightedSampling` *method*), 161
- `eraseTarget()` (`pyAgrum.MonteCarloSampling` *method*), 126
- `eraseTarget()` (`pyAgrum.ShaferShenoyInference` *method*), 100
- `eraseTarget()` (`pyAgrum.ShaferShenoyMNIInference` *method*), 220
- `eraseTarget()` (`pyAgrum.VariableElimination` *method*), 106
- `eraseTarget()` (`pyAgrum.WeightedSampling` *method*), 133
- `eraseTicks()` (`pyAgrum.DiscretizedVariable` *method*), 31
- `eraseValue()` (`pyAgrum.IntegerVariable` *method*), 34
- `eraseValues()` (`pyAgrum.IntegerVariable` *method*), 34
- `errorCallStack()` (`pyAgrum.GumException` *method*), 287
- `errorContent()` (`pyAgrum.GumException` *method*), 287
- `errorType()` (`pyAgrum.GumException` *method*), 287
- `EssentialGraph` (*class in* `pyAgrum`), 79
- `eval()` (`pyAgrum.causal.ASTBinaryOp` *method*), 240
- `eval()` (`pyAgrum.causal.ASTdiv` *method*), 242
- `eval()` (`pyAgrum.causal.ASTjointProba` *method*), 245
- `eval()` (`pyAgrum.causal.ASTminus` *method*), 241
- `eval()` (`pyAgrum.causal.ASTmult` *method*), 243
- `eval()` (`pyAgrum.causal.ASTplus` *method*), 240
- `eval()` (`pyAgrum.causal.ASTposteriorProba` *method*), 246
- `eval()` (`pyAgrum.causal.ASTsum` *method*), 244
- `eval()` (`pyAgrum.causal.ASTtree` *method*), 239
- `eval()` (`pyAgrum.causal.CausalFormula` *method*), 236
- `evidenceImpact()` (`pyAgrum.GibbsSampling` *method*), 119
- `evidenceImpact()` (`pyAgrum.ImportanceSampling` *method*), 140
- `evidenceImpact()` (`pyAgrum.LazyPropagation` *method*), 93
- `evidenceImpact()` (`pyAgrum.LoopyBeliefPropagation` *method*),

- 113
- `evidenceImpact()` (*pyAgrum.LoopyGibbsSampling method*), 147
- `evidenceImpact()` (*pyAgrum.LoopyImportanceSampling method*), 168
- `evidenceImpact()` (*pyAgrum.LoopyMonteCarloSampling method*), 154
- `evidenceImpact()` (*pyAgrum.LoopyWeightedSampling method*), 161
- `evidenceImpact()` (*pyAgrum.MonteCarloSampling method*), 127
- `evidenceImpact()` (*pyAgrum.ShaferShenoyInference method*), 100
- `evidenceImpact()` (*pyAgrum.ShaferShenoyMNIInference method*), 220
- `evidenceImpact()` (*pyAgrum.VariableElimination method*), 106
- `evidenceImpact()` (*pyAgrum.WeightedSampling method*), 133
- `evidenceJointImpact()` (*pyAgrum.LazyPropagation method*), 93
- `evidenceJointImpact()` (*pyAgrum.ShaferShenoyInference method*), 100
- `evidenceJointImpact()` (*pyAgrum.ShaferShenoyMNIInference method*), 221
- `evidenceJointImpact()` (*pyAgrum.VariableElimination method*), 107
- `evidenceProbability()` (*pyAgrum.LazyPropagation method*), 93
- `evidenceProbability()` (*pyAgrum.ShaferShenoyInference method*), 100
- `evidenceProbability()` (*pyAgrum.ShaferShenoyMNIInference method*), 221
- `ExactBNdistance` (*class in pyAgrum*), 73
- `exists()` (*pyAgrum.BayesNet method*), 65
- `exists()` (*pyAgrum.BayesNetFragment method*), 84
- `exists()` (*pyAgrum.InfluenceDiagram method*), 187
- `exists()` (*pyAgrum.MarkovNet method*), 214
- `existsArc()` (*pyAgrum.BayesNet method*), 65
- `existsArc()` (*pyAgrum.BayesNetFragment method*), 84
- `existsArc()` (*pyAgrum.causal.CausalModel method*), 235
- `existsArc()` (*pyAgrum.DAG method*), 9
- `existsArc()` (*pyAgrum.DiGraph method*), 6
- `existsArc()` (*pyAgrum.InfluenceDiagram method*), 187
- `existsArc()` (*pyAgrum.MixedGraph method*), 21
- `existsEdge()` (*pyAgrum.CliqueGraph method*), 16
- `existsEdge()` (*pyAgrum.MarkovNet method*), 214
- `existsEdge()` (*pyAgrum.MixedGraph method*), 21
- `existsEdge()` (*pyAgrum.UndiGraph method*), 12
- `existsNode()` (*pyAgrum.CliqueGraph method*), 16
- `existsNode()` (*pyAgrum.DAG method*), 10
- `existsNode()` (*pyAgrum.DiGraph method*), 6
- `existsNode()` (*pyAgrum.MixedGraph method*), 21
- `existsNode()` (*pyAgrum.UndiGraph method*), 13
- `existsPathBetween()` (*pyAgrum.InfluenceDiagram method*), 187
- `export()` (*in module pyAgrum.lib.image*), 265
- `exportInference()` (*in module pyAgrum.lib.image*), 265
- `extract()` (*pyAgrum.Potential method*), 49
- ## F
- `factor()` (*pyAgrum.MarkovNet method*), 214
- `factors()` (*pyAgrum.MarkovNet method*), 214
- `family()` (*pyAgrum.BayesNet method*), 65
- `family()` (*pyAgrum.BayesNetFragment method*), 84
- `family()` (*pyAgrum.InfluenceDiagram method*), 188
- `fastBN()` (*in module pyAgrum*), 278
- `fastID()` (*in module pyAgrum*), 278
- `fastMN()` (*in module pyAgrum*), 278
- `fastPrototype()` (*pyAgrum.BayesNet static method*), 65
- `fastPrototype()` (*pyAgrum.InfluenceDiagram static method*), 188
- `fastPrototype()` (*pyAgrum.MarkovNet static method*), 214
- `fastToLatex()` (*pyAgrum.causal.ASTBinaryOp method*), 240
- `fastToLatex()` (*pyAgrum.causal.ASTdiv method*), 242
- `fastToLatex()` (*pyAgrum.causal.ASTjointProba method*), 245
- `fastToLatex()` (*pyAgrum.causal.ASTminus method*), 241
- `fastToLatex()` (*pyAgrum.causal.ASTmult method*), 243
- `fastToLatex()` (*pyAgrum.causal.ASTplus method*), 241
- `fastToLatex()` (*pyAgrum.causal.ASTposteriorProba method*), 246
- `fastToLatex()` (*pyAgrum.causal.ASTsum method*), 244
- `fastToLatex()` (*pyAgrum.causal.ASTtree method*), 239
- `FatalError`, 288
- `filename()` (*pyAgrum.SyntaxError method*), 291
- `fillConstraint()` (*pyAgrum.CredalNet method*), 199
- `fillConstraints()` (*pyAgrum.CredalNet method*), 200
- `fillWith()` (*pyAgrum.Potential method*), 49
- `fillWithFunction()` (*pyAgrum.Potential method*), 49
- `findAll()` (*pyAgrum.Potential method*), 50
- `first()` (*pyAgrum.Arc method*), 3



`first()` (*pyAgrum.Edge* method), 4  
`fit()` (*pyAgrum.skbn.BNClassifier* method), 251  
`FormatNotFound`, 288  
`fromBN()` (*pyAgrum.MarkovNet* static method), 215  
`fromdict()` (*pyAgrum.Instantiation* method), 44  
`fromTrainedModel()` (*pyAgrum.skbn.BNClassifier* method), 251  
`frontDoor()` (*pyAgrum.causal.CausalModel* method), 235

## G

`G2()` (*pyAgrum.BN Learner* method), 173  
`generateCPT()` (*pyAgrum.BayesNet* method), 66  
`generateCPTs()` (*pyAgrum.BayesNet* method), 66  
`generateFactor()` (*pyAgrum.MarkovNet* method), 215  
`generateFactors()` (*pyAgrum.MarkovNet* method), 215  
`get()` (*pyAgrum.Potential* method), 50  
`get()` (*pyAgrum.PyAgrumConfiguration* method), 293  
`get_binaryCPT_max()` (*pyAgrum.CredalNet* method), 200  
`get_binaryCPT_min()` (*pyAgrum.CredalNet* method), 200  
`get_params()` (*pyAgrum.skbn.BNClassifier* method), 252  
`getAlltheSystems()` (*pyAgrum.PRMexplorer* method), 230  
`getBN()` (*in module pyAgrum.lib.notebook*), 258  
`getCausalImpact()` (*in module pyAgrum.causal.notebook*), 247  
`getCausalModel()` (*in module pyAgrum.causal.notebook*), 247  
`getCN()` (*in module pyAgrum.lib.notebook*), 260  
`getDecisionGraph()` (*pyAgrum.InfluenceDiagram* method), 189  
`getDirectSubClass()` (*pyAgrum.PRMexplorer* method), 228  
`getDirectSubInterfaces()` (*pyAgrum.PRMexplorer* method), 228  
`getDirectSubTypes()` (*pyAgrum.PRMexplorer* method), 229  
`getDot()` (*in module pyAgrum.lib.notebook*), 263  
`getGraph()` (*in module pyAgrum.lib.notebook*), 263  
`getImplementations()` (*pyAgrum.PRMexplorer* method), 229  
`getInference()` (*in module pyAgrum.lib.notebook*), 261  
`getInfluenceDiagram()` (*in module pyAgrum.lib.notebook*), 259  
`getInformation()` (*in module pyAgrum.lib.explain*), 267  
`getJunctionTree()` (*in module pyAgrum.lib.notebook*), 262  
`getLabelMap()` (*pyAgrum.PRMexplorer* method), 229  
`getLabels()` (*pyAgrum.PRMexplorer* method), 229  
`getMaxNumberOfThreads()` (*in module pyAgrum*), 285

`getMN()` (*in module pyAgrum.lib.notebook*), 259  
`getNumberOfLogicalProcessors()` (*in module pyAgrum*), 285  
`getPosterior()` (*in module pyAgrum*), 277  
`getPosterior()` (*in module pyAgrum.lib.notebook*), 262  
`getPotential()` (*in module pyAgrum.lib.notebook*), 262  
`getSuperClass()` (*pyAgrum.PRMexplorer* method), 229  
`getSuperInterface()` (*pyAgrum.PRMexplorer* method), 229  
`getSuperType()` (*pyAgrum.PRMexplorer* method), 229  
`GibbsBNdistance` (*class in pyAgrum*), 74  
`GibbsSampling` (*class in pyAgrum*), 117  
`graph()` (*pyAgrum.MarkovNet* method), 215  
`GraphError`, 288  
`grep()` (*pyAgrum.PyAgrumConfiguration* method), 293  
`GumException`, 287

## H

`HC()` (*pyAgrum.GibbsSampling* method), 117  
`HC()` (*pyAgrum.ImportanceSampling* method), 138  
`HC()` (*pyAgrum.LazyPropagation* method), 90  
`HC()` (*pyAgrum.LoopyBeliefPropagation* method), 111  
`HC()` (*pyAgrum.LoopyGibbsSampling* method), 145  
`HC()` (*pyAgrum.LoopyImportanceSampling* method), 166  
`HC()` (*pyAgrum.LoopyMonteCarloSampling* method), 152  
`HC()` (*pyAgrum.LoopyWeightedSampling* method), 159  
`HC()` (*pyAgrum.MonteCarloSampling* method), 124  
`HC()` (*pyAgrum.ShaferShenoyInference* method), 97  
`HC()` (*pyAgrum.ShaferShenoyMNIInference* method), 218  
`HC()` (*pyAgrum.VariableElimination* method), 104  
`HC()` (*pyAgrum.WeightedSampling* method), 131  
`hamming()` (*pyAgrum.Instantiation* method), 44  
`hardEvidenceNodes()` (*pyAgrum.GibbsSampling* method), 120  
`hardEvidenceNodes()` (*pyAgrum.ImportanceSampling* method), 140  
`hardEvidenceNodes()` (*pyAgrum.LazyPropagation* method), 94  
`hardEvidenceNodes()` (*pyAgrum.LoopyBeliefPropagation* method), 113  
`hardEvidenceNodes()` (*pyAgrum.LoopyGibbsSampling* method), 147  
`hardEvidenceNodes()` (*pyAgrum.LoopyImportanceSampling* method), 168  
`hardEvidenceNodes()` (*pyAgrum.LoopyMonteCarloSampling* method), 154  
`hardEvidenceNodes()` (*pyAgrum.LoopyWeightedSampling* method),



- 161  
`hardEvidenceNodes()` (*pyAgrum.MonteCarloSampling* method), 127  
`hardEvidenceNodes()` (*pyAgrum.ShaferShenoyInference* method), 101  
`hardEvidenceNodes()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 193  
`hardEvidenceNodes()` (*pyAgrum.ShaferShenoyMNIInference* method), 221  
`hardEvidenceNodes()` (*pyAgrum.VariableElimination* method), 107  
`hardEvidenceNodes()` (*pyAgrum.WeightedSampling* method), 133  
`hasComputedBinaryCPTMinMax()` (*pyAgrum.CredalNet* method), 200  
`hasDirectedPath()` (*pyAgrum.DAG* method), 10  
`hasDirectedPath()` (*pyAgrum.DiGraph* method), 7  
`hasDirectedPath()` (*pyAgrum.MixedGraph* method), 21  
`hasEvidence()` (*pyAgrum.GibbsSampling* method), 120  
`hasEvidence()` (*pyAgrum.ImportanceSampling* method), 140  
`hasEvidence()` (*pyAgrum.LazyPropagation* method), 94  
`hasEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 113  
`hasEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 147  
`hasEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 168  
`hasEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 155  
`hasEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 161  
`hasEvidence()` (*pyAgrum.MonteCarloSampling* method), 127  
`hasEvidence()` (*pyAgrum.ShaferShenoyInference* method), 101  
`hasEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 193  
`hasEvidence()` (*pyAgrum.ShaferShenoyMNIInference* method), 221  
`hasEvidence()` (*pyAgrum.VariableElimination* method), 107  
`hasEvidence()` (*pyAgrum.WeightedSampling* method), 134  
`hasHardEvidence()` (*pyAgrum.GibbsSampling* method), 120  
`hasHardEvidence()` (*pyAgrum.ImportanceSampling* method), 141  
`hasHardEvidence()` (*pyAgrum.LazyPropagation* method), 94  
`hasHardEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 113  
`hasHardEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 147  
`hasHardEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 169  
`hasHardEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 155  
`hasHardEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 162  
`hasHardEvidence()` (*pyAgrum.MonteCarloSampling* method), 127  
`hasHardEvidence()` (*pyAgrum.ShaferShenoyInference* method), 101  
`hasHardEvidence()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 193  
`hasHardEvidence()` (*pyAgrum.ShaferShenoyMNIInference* method), 221  
`hasHardEvidence()` (*pyAgrum.VariableElimination* method), 107  
`hasHardEvidence()` (*pyAgrum.WeightedSampling* method), 134  
`hasMissingValues()` (*pyAgrum.BN Learner* method), 175  
`hasNoForgettingAssumption()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 193  
`hasRunningIntersection()` (*pyAgrum.CliqueGraph* method), 17  
`hasSameStructure()` (*pyAgrum.BayesNet* method), 66  
`hasSameStructure()` (*pyAgrum.BayesNetFragment* method), 84  
`hasSameStructure()` (*pyAgrum.InfluenceDiagram* method), 189  
`hasSameStructure()` (*pyAgrum.MarkovBlanket* method), 81  
`hasSameStructure()` (*pyAgrum.MarkovNet* method), 215  
`hasSoftEvidence()` (*pyAgrum.GibbsSampling* method), 120  
`hasSoftEvidence()` (*pyAgrum.ImportanceSampling* method), 141  
`hasSoftEvidence()` (*pyAgrum.LazyPropagation* method), 94  
`hasSoftEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 113  
`hasSoftEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 148  
`hasSoftEvidence()` (*pyAgrum.LazyPropagation* method), 94

- grum.LoopyImportanceSampling* method), 169
- hasSoftEvidence()* (*pyAgrum.LoopyMonteCarloSampling* method), 155
- hasSoftEvidence()* (*pyAgrum.LoopyWeightedSampling* method), 162
- hasSoftEvidence()* (*pyAgrum.MonteCarloSampling* method), 127
- hasSoftEvidence()* (*pyAgrum.ShaferShenoyInference* method), 101
- hasSoftEvidence()* (*pyAgrum.ShaferShenoyLIMIDInference* method), 193
- hasSoftEvidence()* (*pyAgrum.ShaferShenoyMNIInference* method), 222
- hasSoftEvidence()* (*pyAgrum.VariableElimination* method), 107
- hasSoftEvidence()* (*pyAgrum.WeightedSampling* method), 134
- hasUndirectedCycle()* (*pyAgrum.CliqueGraph* method), 17
- hasUndirectedCycle()* (*pyAgrum.MixedGraph* method), 21
- hasUndirectedCycle()* (*pyAgrum.UndiGraph* method), 13
- head()* (*pyAgrum.Arc* method), 3
- HedgeException* (class in *pyAgrum.causal*), 246
- history()* (*pyAgrum.BN Learner* method), 175
- history()* (*pyAgrum.CN LoopyPropagation* method), 206
- history()* (*pyAgrum.CN MonteCarloSampling* method), 203
- history()* (*pyAgrum.Gibbs BN distance* method), 75
- history()* (*pyAgrum.Gibbs Sampling* method), 120
- history()* (*pyAgrum.ImportanceSampling* method), 141
- history()* (*pyAgrum.Loopy BeliefPropagation* method), 114
- history()* (*pyAgrum.Loopy Gibbs Sampling* method), 148
- history()* (*pyAgrum.Loopy ImportanceSampling* method), 169
- history()* (*pyAgrum.Loopy MonteCarloSampling* method), 155
- history()* (*pyAgrum.Loopy WeightedSampling* method), 162
- history()* (*pyAgrum.MonteCarloSampling* method), 127
- history()* (*pyAgrum.WeightedSampling* method), 134
- |
- I()* (*pyAgrum.LazyPropagation* method), 90
- I()* (*pyAgrum.ShaferShenoyInference* method), 97
- I()* (*pyAgrum.ShaferShenoyMNIInference* method), 218
- identifyingIntervention()* (in module *pyAgrum.causal*), 238
- idFromName()* (*pyAgrum.BayesNet* method), 66
- idFromName()* (*pyAgrum.BayesNetFragment* method), 84
- idFromName()* (*pyAgrum.BN Learner* method), 175
- idFromName()* (*pyAgrum.causal.CausalModel* method), 235
- idFromName()* (*pyAgrum.InfluenceDiagram* method), 189
- idFromName()* (*pyAgrum.MarkovNet* method), 215
- idmLearning()* (*pyAgrum.CredalNet* method), 200
- ids()* (*pyAgrum.BayesNet* method), 66
- ids()* (*pyAgrum.BayesNetFragment* method), 84
- ids()* (*pyAgrum.InfluenceDiagram* method), 189
- ids()* (*pyAgrum.MarkovNet* method), 215
- ImportanceSampling* (class in *pyAgrum*), 138
- inc()* (*pyAgrum.Instantiation* method), 44
- incIn()* (*pyAgrum.Instantiation* method), 44
- incNotVar()* (*pyAgrum.Instantiation* method), 45
- incOut()* (*pyAgrum.Instantiation* method), 45
- incVar()* (*pyAgrum.Instantiation* method), 45
- independenceListForPairs()* (in module *pyAgrum.lib.explain*), 267
- index()* (*pyAgrum.DiscreteVariable* method), 25
- index()* (*pyAgrum.DiscretizedVariable* method), 31
- index()* (*pyAgrum.IntegerVariable* method), 34
- index()* (*pyAgrum.LabelizedVariable* method), 28
- index()* (*pyAgrum.RangeVariable* method), 37
- inferenceType()* (*pyAgrum.CN LoopyPropagation* method), 206
- InferenceType\_nodeToNeighbours* (*pyAgrum.CN LoopyPropagation* attribute), 205
- InferenceType\_ordered* (*pyAgrum.CN LoopyPropagation* attribute), 206
- InferenceType\_randomOrder* (*pyAgrum.CN LoopyPropagation* attribute), 206
- InfluenceDiagram* (class in *pyAgrum*), 184
- influenceDiagram()* (*pyAgrum.ShaferShenoyLIMIDInference* method), 193
- initApproximationScheme()* (*pyAgrum.Gibbs BN distance* method), 75
- initRandom()* (in module *pyAgrum*), 284
- inOverflow()* (*pyAgrum.Instantiation* method), 44
- insertEvidenceFile()* (*pyAgrum.CN LoopyPropagation* method), 206
- insertEvidenceFile()* (*pyAgrum.CN MonteCarloSampling* method), 203
- insertModalsFile()* (*pyAgrum.CN LoopyPropagation* method), 206
- insertModalsFile()* (*pyAgrum.CN MonteCarloSampling* method), 203

- installAscendants() (*pyAgrum.BayesNetFragment method*), 85
- installCPT() (*pyAgrum.BayesNetFragment method*), 85
- installMarginal() (*pyAgrum.BayesNetFragment method*), 85
- installNode() (*pyAgrum.BayesNetFragment method*), 85
- Instantiation (*class in pyAgrum*), 42
- instantiation() (*pyAgrum.CredalNet method*), 200
- integerDomain() (*pyAgrum.IntegerVariable method*), 34
- IntegerVariable (*class in pyAgrum*), 33
- interAttributes() (*pyAgrum.PRMexplorer method*), 230
- interfaces() (*pyAgrum.PRMexplorer method*), 230
- interReferences() (*pyAgrum.PRMexplorer method*), 230
- intervalToCredal() (*pyAgrum.CredalNet method*), 201
- intervalToCredalWithFiles() (*pyAgrum.CredalNet method*), 201
- InvalidArc, 288
- InvalidArgument, 288
- InvalidArgumentsNumber, 289
- InvalidDirectedCycle, 289
- InvalidEdge, 289
- InvalidNode, 289
- inverse() (*pyAgrum.Potential method*), 50
- IOError, 288
- isAttribute() (*pyAgrum.PRMexplorer method*), 230
- isChanceNode() (*pyAgrum.InfluenceDiagram method*), 189
- isClass() (*pyAgrum.PRMexplorer method*), 230
- isDecisionNode() (*pyAgrum.InfluenceDiagram method*), 189
- isDrawnAtRandom() (*pyAgrum.GibbsBNdistance method*), 75
- isDrawnAtRandom() (*pyAgrum.GibbsSampling method*), 120
- isDrawnAtRandom() (*pyAgrum.LoopyGibbsSampling method*), 148
- isEnabledEpsilon() (*pyAgrum.GibbsBNdistance method*), 75
- isEnabledMaxIter() (*pyAgrum.GibbsBNdistance method*), 75
- isEnabledMaxTime() (*pyAgrum.GibbsBNdistance method*), 75
- isEnabledMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 76
- isIndependent() (*pyAgrum.BayesNet method*), 66
- isIndependent() (*pyAgrum.BayesNetFragment method*), 85
- isIndependent() (*pyAgrum.InfluenceDiagram method*), 189
- isIndependent() (*pyAgrum.MarkovNet method*), 215
- isInstalledNode() (*pyAgrum.BayesNetFragment method*), 85
- isInterface() (*pyAgrum.PRMexplorer method*), 230
- isJoinTree() (*pyAgrum.CliqueGraph method*), 17
- isJointTarget() (*pyAgrum.LazyPropagation method*), 94
- isJointTarget() (*pyAgrum.ShaferShenoyInference method*), 101
- isJointTarget() (*pyAgrum.ShaferShenoyMNIInference method*), 222
- isJointTarget() (*pyAgrum.VariableElimination method*), 108
- isLabel() (*pyAgrum.LabelizedVariable method*), 29
- isMutable() (*pyAgrum.Instantiation method*), 45
- isNonZeroMap() (*pyAgrum.Potential method*), 50
- isOMP() (*in module pyAgrum*), 285
- isSeparatelySpecified() (*pyAgrum.CredalNet method*), 201
- isSolvable() (*pyAgrum.ShaferShenoyLIMIDInference method*), 193
- isTarget() (*pyAgrum.GibbsSampling method*), 121
- isTarget() (*pyAgrum.ImportanceSampling method*), 141
- isTarget() (*pyAgrum.LazyPropagation method*), 94
- isTarget() (*pyAgrum.LoopyBeliefPropagation method*), 114
- isTarget() (*pyAgrum.LoopyGibbsSampling method*), 148
- isTarget() (*pyAgrum.LoopyImportanceSampling method*), 169
- isTarget() (*pyAgrum.LoopyMonteCarloSampling method*), 155
- isTarget() (*pyAgrum.LoopyWeightedSampling method*), 162
- isTarget() (*pyAgrum.MonteCarloSampling method*), 128
- isTarget() (*pyAgrum.ShaferShenoyInference method*), 101
- isTarget() (*pyAgrum.ShaferShenoyMNIInference method*), 222
- isTarget() (*pyAgrum.VariableElimination method*), 108
- isTarget() (*pyAgrum.WeightedSampling method*), 134
- isTick() (*pyAgrum.DiscretizedVariable method*), 32
- isType() (*pyAgrum.PRMexplorer method*), 231
- isUtilityNode() (*pyAgrum.InfluenceDiagram method*), 189
- ## J
- jointMutualInformation() (*pyAgrum.LazyPropagation method*), 95
- jointMutualInformation() (*pyAgrum.ShaferShenoyInference method*), 102
- jointMutualInformation() (*pyAgrum.ShaferShenoyMNIInference method*), 222

- `jointMutualInformation()` (`pyAgrum.VariableElimination` method), 108
- `jointPosterior()` (`pyAgrum.LazyPropagation` method), 95
- `jointPosterior()` (`pyAgrum.ShaferShenoyInference` method), 102
- `jointPosterior()` (`pyAgrum.ShaferShenoyMNIInference` method), 222
- `jointPosterior()` (`pyAgrum.VariableElimination` method), 108
- `jointProbability()` (`pyAgrum.BayesNet` method), 66
- `jointProbability()` (`pyAgrum.BayesNetFragment` method), 85
- `jointTree()` (`pyAgrum.LazyPropagation` method), 95
- `jointTree()` (`pyAgrum.ShaferShenoyInference` method), 102
- `jointTree()` (`pyAgrum.ShaferShenoyMNIInference` method), 222
- `jointTargets()` (`pyAgrum.LazyPropagation` method), 95
- `jointTargets()` (`pyAgrum.ShaferShenoyInference` method), 102
- `jointTargets()` (`pyAgrum.ShaferShenoyMNIInference` method), 223
- `jointTargets()` (`pyAgrum.VariableElimination` method), 108
- `junctionTree()` (`pyAgrum.JunctionTreeGenerator` method), 79
- `junctionTree()` (`pyAgrum.LazyPropagation` method), 95
- `junctionTree()` (`pyAgrum.ShaferShenoyInference` method), 102
- `junctionTree()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 193
- `junctionTree()` (`pyAgrum.ShaferShenoyMNIInference` method), 223
- `junctionTree()` (`pyAgrum.VariableElimination` method), 109
- `JunctionTreeGenerator` (class in `pyAgrum`), 78
- K**
- `KL()` (`pyAgrum.Potential` method), 48
- `knw` (`pyAgrum.causal.ASTposteriorProba` property), 246
- L**
- `label()` (`pyAgrum.DiscreteVariable` method), 25
- `label()` (`pyAgrum.DiscretizedVariable` method), 32
- `label()` (`pyAgrum.IntegerVariable` method), 34
- `label()` (`pyAgrum.LabelizedVariable` method), 29
- `label()` (`pyAgrum.RangeVariable` method), 37
- `LabelizedVariable` (class in `pyAgrum`), 27
- `labels()` (`pyAgrum.DiscreteVariable` method), 26
- `labels()` (`pyAgrum.DiscretizedVariable` method), 32
- `labels()` (`pyAgrum.IntegerVariable` method), 34
- `labels()` (`pyAgrum.LabelizedVariable` method), 29
- `labels()` (`pyAgrum.RangeVariable` method), 37
- `lagrangeNormalization()` (`pyAgrum.CredalNet` method), 201
- `latentVariables()` (`pyAgrum.BNLearner` method), 175
- `latentVariablesIds()` (`pyAgrum.causal.CausalModel` method), 235
- `latexQuery()` (`pyAgrum.causal.CausalFormula` method), 237
- `LazyPropagation` (class in `pyAgrum`), 90
- `learnBN()` (`pyAgrum.BNLearner` method), 175
- `learnDAG()` (`pyAgrum.BNLearner` method), 176
- `learnMixedStructure()` (`pyAgrum.BNLearner` method), 176
- `learnParameters()` (`pyAgrum.BNLearner` method), 176
- `line()` (`pyAgrum.SyntaxError` method), 291
- `load()` (`pyAgrum.PRMexplorer` method), 231
- `load()` (`pyAgrum.PyAgrumConfiguration` method), 293
- `loadBIF()` (`pyAgrum.BayesNet` method), 67
- `loadBIFXML()` (`pyAgrum.BayesNet` method), 67
- `loadBIFXML()` (`pyAgrum.InfluenceDiagram` method), 189
- `loadBN()` (in module `pyAgrum`), 279
- `loadDSL()` (`pyAgrum.BayesNet` method), 67
- `loadID()` (in module `pyAgrum`), 281
- `loadMN()` (in module `pyAgrum`), 280
- `loadNET()` (`pyAgrum.BayesNet` method), 67
- `loadO3PRM()` (`pyAgrum.BayesNet` method), 68
- `loadUAI()` (`pyAgrum.BayesNet` method), 68
- `loadUAI()` (`pyAgrum.MarkovNet` method), 215
- `log10DomainSize()` (`pyAgrum.BayesNet` method), 68
- `log10DomainSize()` (`pyAgrum.BayesNetFragment` method), 86
- `log10DomainSize()` (`pyAgrum.InfluenceDiagram` method), 189
- `log10DomainSize()` (`pyAgrum.MarkovNet` method), 216
- `log2()` (`pyAgrum.Potential` method), 50
- `log2JointProbability()` (`pyAgrum.BayesNet` method), 68
- `log2JointProbability()` (`pyAgrum.BayesNetFragment` method), 86
- `log2likelihood()` (`pyAgrum.BNDatabaseGenerator` method), 72
- `logLikelihood()` (`pyAgrum.BNLearner` method), 176
- `loopIn()` (`pyAgrum.Potential` method), 51
- `LoopyBeliefPropagation` (class in `pyAgrum`), 110
- `LoopyGibbsSampling` (class in `pyAgrum`), 144
- `LoopyImportanceSampling` (class in `pyAgrum`), 166
- `LoopyMonteCarloSampling` (class in `pyAgrum`), 152
- `LoopyWeightedSampling` (class in `pyAgrum`), 159



## M

- `makeInference()` (*pyAgrum.CNLoopyPropagation method*), 207
- `makeInference()` (*pyAgrum.CNMonteCarloSampling method*), 203
- `makeInference()` (*pyAgrum.GibbsSampling method*), 121
- `makeInference()` (*pyAgrum.ImportanceSampling method*), 141
- `makeInference()` (*pyAgrum.LazyPropagation method*), 95
- `makeInference()` (*pyAgrum.LoopyBeliefPropagation method*), 114
- `makeInference()` (*pyAgrum.LoopyGibbsSampling method*), 148
- `makeInference()` (*pyAgrum.LoopyImportanceSampling method*), 169
- `makeInference()` (*pyAgrum.LoopyMonteCarloSampling method*), 156
- `makeInference()` (*pyAgrum.LoopyWeightedSampling method*), 162
- `makeInference()` (*pyAgrum.MonteCarloSampling method*), 128
- `makeInference()` (*pyAgrum.ShaferShenoyInference method*), 102
- `makeInference()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 193
- `makeInference()` (*pyAgrum.ShaferShenoyMNInference method*), 223
- `makeInference()` (*pyAgrum.VariableElimination method*), 109
- `makeInference()` (*pyAgrum.WeightedSampling method*), 135
- `makeInference_()` (*pyAgrum.LoopyGibbsSampling method*), 148
- `makeInference_()` (*pyAgrum.LoopyImportanceSampling method*), 170
- `makeInference_()` (*pyAgrum.LoopyMonteCarloSampling method*), 156
- `makeInference_()` (*pyAgrum.LoopyWeightedSampling method*), 163
- `marginalMax()` (*pyAgrum.CNLoopyPropagation method*), 207
- `marginalMax()` (*pyAgrum.CNMonteCarloSampling method*), 203
- `marginalMin()` (*pyAgrum.CNLoopyPropagation method*), 207
- `marginalMin()` (*pyAgrum.CNMonteCarloSampling method*), 204
- `margMaxIn()` (*pyAgrum.Potential method*), 51
- `margMaxOut()` (*pyAgrum.Potential method*), 51
- `margMinIn()` (*pyAgrum.Potential method*), 51
- `margMinOut()` (*pyAgrum.Potential method*), 51
- `margProdIn()` (*pyAgrum.Potential method*), 51
- `margProdOut()` (*pyAgrum.Potential method*), 51
- `margSumIn()` (*pyAgrum.Potential method*), 52
- `margSumOut()` (*pyAgrum.Potential method*), 52
- `MarkovBlanket` (*class in pyAgrum*), 81
- `MarkovNet` (*class in pyAgrum*), 212
- `max()` (*pyAgrum.Potential method*), 52
- `maxIter()` (*pyAgrum.BN Learner method*), 176
- `maxIter()` (*pyAgrum.CNLoopyPropagation method*), 207
- `maxIter()` (*pyAgrum.CNMonteCarloSampling method*), 204
- `maxIter()` (*pyAgrum.GibbsBNdistance method*), 76
- `maxIter()` (*pyAgrum.GibbsSampling method*), 121
- `maxIter()` (*pyAgrum.ImportanceSampling method*), 142
- `maxIter()` (*pyAgrum.LoopyBeliefPropagation method*), 114
- `maxIter()` (*pyAgrum.LoopyGibbsSampling method*), 149
- `maxIter()` (*pyAgrum.LoopyImportanceSampling method*), 170
- `maxIter()` (*pyAgrum.LoopyMonteCarloSampling method*), 156
- `maxIter()` (*pyAgrum.LoopyWeightedSampling method*), 163
- `maxIter()` (*pyAgrum.MonteCarloSampling method*), 128
- `maxIter()` (*pyAgrum.WeightedSampling method*), 135
- `maxNonOne()` (*pyAgrum.Potential method*), 52
- `maxNonOneParam()` (*pyAgrum.BayesNet method*), 68
- `maxNonOneParam()` (*pyAgrum.BayesNetFragment method*), 86
- `maxNonOneParam()` (*pyAgrum.MarkovNet method*), 216
- `maxParam()` (*pyAgrum.BayesNet method*), 68
- `maxParam()` (*pyAgrum.BayesNetFragment method*), 86
- `maxParam()` (*pyAgrum.MarkovNet method*), 216
- `maxTime()` (*pyAgrum.BN Learner method*), 177
- `maxTime()` (*pyAgrum.CNLoopyPropagation method*), 207
- `maxTime()` (*pyAgrum.CNMonteCarloSampling method*), 204
- `maxTime()` (*pyAgrum.GibbsBNdistance method*), 76
- `maxTime()` (*pyAgrum.GibbsSampling method*), 121
- `maxTime()` (*pyAgrum.ImportanceSampling method*), 142
- `maxTime()` (*pyAgrum.LoopyBeliefPropagation method*), 114
- `maxTime()` (*pyAgrum.LoopyGibbsSampling method*), 149
- `maxTime()` (*pyAgrum.LoopyImportanceSampling method*), 170
- `maxTime()` (*pyAgrum.LoopyMonteCarloSampling method*), 156

- method*), 156
- `maxTime()` (*pyAgrum.LoopyWeightedSampling method*), 163
- `maxTime()` (*pyAgrum.MonteCarloSampling method*), 128
- `maxTime()` (*pyAgrum.WeightedSampling method*), 135
- `maxVal()` (*pyAgrum.RangeVariable method*), 37
- `maxVarDomainSize()` (*pyAgrum.BayesNet method*), 69
- `maxVarDomainSize()` (*pyAgrum.BayesNetFragment method*), 86
- `maxVarDomainSize()` (*pyAgrum.MarkovNet method*), 216
- `meanVar()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 193
- `messageApproximationScheme()` (*pyAgrum.BN Learner method*), 177
- `messageApproximationScheme()` (*pyAgrum.CNLoopyPropagation method*), 207
- `messageApproximationScheme()` (*pyAgrum.CNMonteCarloSampling method*), 204
- `messageApproximationScheme()` (*pyAgrum.GibbsBNdistance method*), 76
- `messageApproximationScheme()` (*pyAgrum.GibbsSampling method*), 121
- `messageApproximationScheme()` (*pyAgrum.ImportanceSampling method*), 142
- `messageApproximationScheme()` (*pyAgrum.LoopyBeliefPropagation method*), 114
- `messageApproximationScheme()` (*pyAgrum.LoopyGibbsSampling method*), 149
- `messageApproximationScheme()` (*pyAgrum.LoopyImportanceSampling method*), 170
- `messageApproximationScheme()` (*pyAgrum.LoopyMonteCarloSampling method*), 156
- `messageApproximationScheme()` (*pyAgrum.LoopyWeightedSampling method*), 163
- `messageApproximationScheme()` (*pyAgrum.MonteCarloSampling method*), 128
- `messageApproximationScheme()` (*pyAgrum.WeightedSampling method*), 135
- `minimalCondSet()` (*pyAgrum.BayesNet method*), 69
- `minimalCondSet()` (*pyAgrum.BayesNetFragment method*), 86
- `minimalCondSet()` (*pyAgrum.MarkovNet method*), 216
- `minNonZero()` (*pyAgrum.Potential method*), 52
- `minNonZeroParam()` (*pyAgrum.BayesNet method*), 69
- `minNonZeroParam()` (*pyAgrum.BayesNetFragment method*), 86
- `minNonZeroParam()` (*pyAgrum.MarkovNet method*), 216
- `minParam()` (*pyAgrum.BayesNet method*), 69
- `minParam()` (*pyAgrum.BayesNetFragment method*), 86
- `minParam()` (*pyAgrum.MarkovNet method*), 216
- `minVal()` (*pyAgrum.RangeVariable method*), 37
- `MixedGraph` (class in *pyAgrum*), 18
- `mixedGraph()` (*pyAgrum.EssentialGraph method*), 80
- `mixedOrientedPath()` (*pyAgrum.MixedGraph method*), 22
- `mixedUnorientedPath()` (*pyAgrum.MixedGraph method*), 22
- `MN()` (*pyAgrum.ShaferShenoyMNInference method*), 218
- module
- pyAgrum.causal.notebook*, 247
  - MonteCarloSampling* (class in *pyAgrum*), 124
  - moralGraph()* (*pyAgrum.BayesNet method*), 69
  - moralGraph()* (*pyAgrum.BayesNetFragment method*), 86
  - moralGraph()* (*pyAgrum.DAG method*), 10
  - moralGraph()* (*pyAgrum.InfluenceDiagram method*), 190
  - moralizedAncestralGraph()* (*pyAgrum.BayesNet method*), 69
  - moralizedAncestralGraph()* (*pyAgrum.BayesNetFragment method*), 87
- `minEpsilonRate()` (*pyAgrum.GibbsSampling method*), 121
- `minEpsilonRate()` (*pyAgrum.ImportanceSampling method*), 142
- `minEpsilonRate()` (*pyAgrum.LoopyBeliefPropagation method*), 114
- `minEpsilonRate()` (*pyAgrum.LoopyGibbsSampling method*), 149
- `minEpsilonRate()` (*pyAgrum.LoopyImportanceSampling method*), 170
- `minEpsilonRate()` (*pyAgrum.LoopyMonteCarloSampling method*), 156
- `minEpsilonRate()` (*pyAgrum.LoopyWeightedSampling method*), 163
- `minEpsilonRate()` (*pyAgrum.MonteCarloSampling method*), 128
- `minEpsilonRate()` (*pyAgrum.WeightedSampling method*), 135
- `minimalCondSet()` (*pyAgrum.BayesNet method*), 69
- `minimalCondSet()` (*pyAgrum.BayesNetFragment method*), 86
- `minimalCondSet()` (*pyAgrum.MarkovNet method*), 216
- `minNonZero()` (*pyAgrum.Potential method*), 52
- `minNonZeroParam()` (*pyAgrum.BayesNet method*), 69
- `minNonZeroParam()` (*pyAgrum.BayesNetFragment method*), 86
- `minNonZeroParam()` (*pyAgrum.MarkovNet method*), 216
- `minParam()` (*pyAgrum.BayesNet method*), 69
- `minParam()` (*pyAgrum.BayesNetFragment method*), 86
- `minParam()` (*pyAgrum.MarkovNet method*), 216
- `minVal()` (*pyAgrum.RangeVariable method*), 37
- `MixedGraph` (class in *pyAgrum*), 18
- `mixedGraph()` (*pyAgrum.EssentialGraph method*), 80
- `mixedOrientedPath()` (*pyAgrum.MixedGraph method*), 22
- `mixedUnorientedPath()` (*pyAgrum.MixedGraph method*), 22
- `MN()` (*pyAgrum.ShaferShenoyMNInference method*), 218
- module
- pyAgrum.causal.notebook*, 247
  - MonteCarloSampling* (class in *pyAgrum*), 124
  - moralGraph()* (*pyAgrum.BayesNet method*), 69
  - moralGraph()* (*pyAgrum.BayesNetFragment method*), 86
  - moralGraph()* (*pyAgrum.DAG method*), 10
  - moralGraph()* (*pyAgrum.InfluenceDiagram method*), 190
  - moralizedAncestralGraph()* (*pyAgrum.BayesNet method*), 69
  - moralizedAncestralGraph()* (*pyAgrum.BayesNetFragment method*), 87
- `minEpsilonRate()` (*pyAgrum.GibbsBNdistance method*), 76

- `moralizedAncestralGraph()` (`pyAgrum.DAG` method), 10
- `moralizedAncestralGraph()` (`pyAgrum.InfluenceDiagram` method), 190
- ## N
- `name()` (`pyAgrum.DiscreteVariable` method), 26
- `name()` (`pyAgrum.DiscretizedVariable` method), 32
- `name()` (`pyAgrum.IntegerVariable` method), 35
- `name()` (`pyAgrum.LabelizedVariable` method), 29
- `name()` (`pyAgrum.RangeVariable` method), 38
- `nameFromId()` (`pyAgrum.BN Learner` method), 177
- `names()` (`pyAgrum.BayesNet` method), 69
- `names()` (`pyAgrum.BayesNetFragment` method), 87
- `names()` (`pyAgrum.BN Learner` method), 177
- `names()` (`pyAgrum.causal.CausalModel` method), 235
- `names()` (`pyAgrum.InfluenceDiagram` method), 190
- `names()` (`pyAgrum.MarkovNet` method), 216
- `nbCols()` (`pyAgrum.BN Learner` method), 177
- `nbrDim()` (`pyAgrum.Instantiation` method), 45
- `nbrDim()` (`pyAgrum.Potential` method), 52
- `nbrDrawnVar()` (`pyAgrum.GibbsBNdistance` method), 76
- `nbrDrawnVar()` (`pyAgrum.GibbsSampling` method), 121
- `nbrDrawnVar()` (`pyAgrum.LoopyGibbsSampling` method), 149
- `nbrEvidence()` (`pyAgrum.GibbsSampling` method), 121
- `nbrEvidence()` (`pyAgrum.ImportanceSampling` method), 142
- `nbrEvidence()` (`pyAgrum.LazyPropagation` method), 95
- `nbrEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 114
- `nbrEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 149
- `nbrEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 170
- `nbrEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 156
- `nbrEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 163
- `nbrEvidence()` (`pyAgrum.MonteCarloSampling` method), 128
- `nbrEvidence()` (`pyAgrum.ShaferShenoyInference` method), 103
- `nbrEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 193
- `nbrEvidence()` (`pyAgrum.ShaferShenoyMNInference` method), 223
- `nbrEvidence()` (`pyAgrum.VariableElimination` method), 109
- `nbrEvidence()` (`pyAgrum.WeightedSampling` method), 135
- `nbrHardEvidence()` (`pyAgrum.GibbsSampling` method), 121
- `nbrHardEvidence()` (`pyAgrum.ImportanceSampling` method), 142
- `nbrHardEvidence()` (`pyAgrum.LazyPropagation` method), 96
- `nbrHardEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 114
- `nbrHardEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 149
- `nbrHardEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 170
- `nbrHardEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 156
- `nbrHardEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 163
- `nbrHardEvidence()` (`pyAgrum.MonteCarloSampling` method), 128
- `nbrHardEvidence()` (`pyAgrum.ShaferShenoyInference` method), 103
- `nbrHardEvidence()` (`pyAgrum.ShaferShenoyLIMIDInference` method), 193
- `nbrHardEvidence()` (`pyAgrum.ShaferShenoyMNInference` method), 223
- `nbrHardEvidence()` (`pyAgrum.VariableElimination` method), 109
- `nbrHardEvidence()` (`pyAgrum.WeightedSampling` method), 135
- `nbrIterations()` (`pyAgrum.BN Learner` method), 177
- `nbrIterations()` (`pyAgrum.CNLoopyPropagation` method), 207
- `nbrIterations()` (`pyAgrum.CNMonteCarloSampling` method), 204
- `nbrIterations()` (`pyAgrum.GibbsBNdistance` method), 76
- `nbrIterations()` (`pyAgrum.GibbsSampling` method), 122
- `nbrIterations()` (`pyAgrum.ImportanceSampling` method), 142
- `nbrIterations()` (`pyAgrum.LoopyBeliefPropagation` method), 115
- `nbrIterations()` (`pyAgrum.LoopyGibbsSampling` method), 149
- `nbrIterations()` (`pyAgrum.LoopyImportanceSampling` method), 170
- `nbrIterations()` (`pyAgrum.LoopyMonteCarloSampling` method), 156

`nbrIterations()` (*pyAgrum.LoopyWeightedSampling method*), 163  
`nbrIterations()` (*pyAgrum.MonteCarloSampling method*), 129  
`nbrIterations()` (*pyAgrum.WeightedSampling method*), 135  
`nbrJointTargets()` (*pyAgrum.LazyPropagation method*), 96  
`nbrJointTargets()` (*pyAgrum.ShaferShenoyInference method*), 103  
`nbrJointTargets()` (*pyAgrum.ShaferShenoyMNIInference method*), 223  
`nbRows()` (*pyAgrum.BN Learner method*), 177  
`nbrSoftEvidence()` (*pyAgrum.GibbsSampling method*), 122  
`nbrSoftEvidence()` (*pyAgrum.ImportanceSampling method*), 142  
`nbrSoftEvidence()` (*pyAgrum.LazyPropagation method*), 96  
`nbrSoftEvidence()` (*pyAgrum.LoopyBeliefPropagation method*), 115  
`nbrSoftEvidence()` (*pyAgrum.LoopyGibbsSampling method*), 149  
`nbrSoftEvidence()` (*pyAgrum.LoopyImportanceSampling method*), 170  
`nbrSoftEvidence()` (*pyAgrum.LoopyMonteCarloSampling method*), 156  
`nbrSoftEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 163  
`nbrSoftEvidence()` (*pyAgrum.MonteCarloSampling method*), 129  
`nbrSoftEvidence()` (*pyAgrum.ShaferShenoyInference method*), 103  
`nbrSoftEvidence()` (*pyAgrum.ShaferShenoyMNIInference method*), 223  
`nbrSoftEvidence()` (*pyAgrum.VariableElimination method*), 109  
`nbrSoftEvidence()` (*pyAgrum.WeightedSampling method*), 135  
`neighbours()` (*pyAgrum.CliqueGraph method*), 17  
`neighbours()` (*pyAgrum.EssentialGraph method*), 80  
`neighbours()` (*pyAgrum.MarkovNet method*), 216  
`neighbours()` (*pyAgrum.MixedGraph method*), 22  
`neighbours()` (*pyAgrum.UndiGraph method*), 13  
`new_abs()` (*pyAgrum.Potential method*), 53  
`new_log2()` (*pyAgrum.Potential method*), 53  
`new_sq()` (*pyAgrum.Potential method*), 53  
`newFactory()` (*pyAgrum.Potential method*), 52  
`NoChild`, 289  
`nodeId()` (*pyAgrum.BayesNet method*), 69  
`nodeId()` (*pyAgrum.BayesNetFragment method*), 87  
`nodeId()` (*pyAgrum.InfluenceDiagram method*), 190  
`nodeId()` (*pyAgrum.MarkovNet method*), 216  
`nodes()` (*pyAgrum.BayesNet method*), 70  
`nodes()` (*pyAgrum.BayesNetFragment method*), 87  
`nodes()` (*pyAgrum.causal.CausalModel method*), 236  
`nodes()` (*pyAgrum.CliqueGraph method*), 17  
`nodes()` (*pyAgrum.DAG method*), 10  
`nodes()` (*pyAgrum.DiGraph method*), 7  
`nodes()` (*pyAgrum.EssentialGraph method*), 80  
`nodes()` (*pyAgrum.InfluenceDiagram method*), 190  
`nodes()` (*pyAgrum.MarkovBlanket method*), 81  
`nodes()` (*pyAgrum.MarkovNet method*), 216  
`nodes()` (*pyAgrum.MixedGraph method*), 22  
`nodes()` (*pyAgrum.UndiGraph method*), 13  
`nodes2ConnectedComponent()` (*pyAgrum.CliqueGraph method*), 17  
`nodes2ConnectedComponent()` (*pyAgrum.MixedGraph method*), 22  
`nodes2ConnectedComponent()` (*pyAgrum.UndiGraph method*), 13  
`nodeset()` (*pyAgrum.BayesNet method*), 70  
`nodeset()` (*pyAgrum.BayesNetFragment method*), 87  
`nodeset()` (*pyAgrum.InfluenceDiagram method*), 190  
`nodeset()` (*pyAgrum.MarkovNet method*), 216  
`nodeType()` (*pyAgrum.CredalNet method*), 201  
`NodeType_Credal` (*pyAgrum.CredalNet attribute*), 197



- NodeType\_Indic (*pyAgrum.CredalNet* attribute), 197  
 NodeType\_Precise (*pyAgrum.CredalNet* attribute), 197  
 NodeType\_Vacuous (*pyAgrum.CredalNet* attribute), 197  
 noising() (*pyAgrum.Potential* method), 53  
 NoNeighbour, 289  
 NoParent, 289  
 normalize() (*pyAgrum.Potential* method), 53  
 normalizeAsCPT() (*pyAgrum.Potential* method), 53  
 NotFound, 290  
 NullElement, 290  
 numerical() (*pyAgrum.DiscreteVariable* method), 26  
 numerical() (*pyAgrum.DiscretizedVariable* method), 32  
 numerical() (*pyAgrum.IntegerVariable* method), 35  
 numerical() (*pyAgrum.LabelizedVariable* method), 29  
 numerical() (*pyAgrum.RangeVariable* method), 38
- ## O
- observationalBN() (*pyAgrum.causal.CausalModel* method), 236  
 op1 (*pyAgrum.causal.ASTBinaryOp* property), 240  
 op1 (*pyAgrum.causal.ASTdiv* property), 242  
 op1 (*pyAgrum.causal.ASTminus* property), 242  
 op1 (*pyAgrum.causal.ASTMult* property), 243  
 op1 (*pyAgrum.causal.ASTplus* property), 241  
 op2 (*pyAgrum.causal.ASTBinaryOp* property), 240  
 op2 (*pyAgrum.causal.ASTdiv* property), 242  
 op2 (*pyAgrum.causal.ASTminus* property), 242  
 op2 (*pyAgrum.causal.ASTMult* property), 243  
 op2 (*pyAgrum.causal.ASTplus* property), 241  
 OperationNotAllowed, 290  
 optimalDecision() (*pyAgrum.ShaferShenoyLIMIDInference* method), 194  
 other() (*pyAgrum.Arc* method), 3  
 other() (*pyAgrum.Edge* method), 4  
 OutOfBounds, 290
- ## P
- parents() (*pyAgrum.BayesNet* method), 70  
 parents() (*pyAgrum.BayesNetFragment* method), 87  
 parents() (*pyAgrum.causal.CausalModel* method), 236  
 parents() (*pyAgrum.DAG* method), 10  
 parents() (*pyAgrum.DiGraph* method), 7  
 parents() (*pyAgrum.EssentialGraph* method), 80  
 parents() (*pyAgrum.InfluenceDiagram* method), 190  
 parents() (*pyAgrum.MarkovBlanket* method), 82  
 parents() (*pyAgrum.MixedGraph* method), 22  
 partialUndiGraph() (*pyAgrum.CliqueGraph* method), 17  
 partialUndiGraph() (*pyAgrum.MixedGraph* method), 22  
 partialUndiGraph() (*pyAgrum.UndiGraph* method), 13  
 periodSize() (*pyAgrum.BNLearner* method), 177  
 periodSize() (*pyAgrum.CNLoopyPropagation* method), 207  
 periodSize() (*pyAgrum.CNMonteCarloSampling* method), 204  
 periodSize() (*pyAgrum.GibbsBNdistance* method), 76  
 periodSize() (*pyAgrum.GibbsSampling* method), 122  
 periodSize() (*pyAgrum.ImportanceSampling* method), 142  
 periodSize() (*pyAgrum.LoopyBeliefPropagation* method), 115  
 periodSize() (*pyAgrum.LoopyGibbsSampling* method), 149  
 periodSize() (*pyAgrum.LoopyImportanceSampling* method), 170  
 periodSize() (*pyAgrum.LoopyMonteCarloSampling* method), 157  
 periodSize() (*pyAgrum.LoopyWeightedSampling* method), 163  
 periodSize() (*pyAgrum.MonteCarloSampling* method), 129  
 periodSize() (*pyAgrum.WeightedSampling* method), 135  
 pos() (*pyAgrum.Instantiation* method), 45  
 pos() (*pyAgrum.Potential* method), 53  
 posLabel() (*pyAgrum.LabelizedVariable* method), 29  
 posterior() (*pyAgrum.GibbsSampling* method), 122  
 posterior() (*pyAgrum.ImportanceSampling* method), 142  
 posterior() (*pyAgrum.LazyPropagation* method), 96  
 posterior() (*pyAgrum.LoopyBeliefPropagation* method), 115  
 posterior() (*pyAgrum.LoopyGibbsSampling* method), 150  
 posterior() (*pyAgrum.LoopyImportanceSampling* method), 171  
 posterior() (*pyAgrum.LoopyMonteCarloSampling* method), 157  
 posterior() (*pyAgrum.LoopyWeightedSampling* method), 164  
 posterior() (*pyAgrum.MonteCarloSampling* method), 129  
 posterior() (*pyAgrum.ShaferShenoyInference* method), 103  
 posterior() (*pyAgrum.ShaferShenoyLIMIDInference* method), 194  
 posterior() (*pyAgrum.ShaferShenoyMNIInference* method), 224  
 posterior() (*pyAgrum.VariableElimination* method), 109  
 posterior() (*pyAgrum.WeightedSampling* method), 136  
 posteriorUtility() (*pyAgrum.ShaferShenoyLIMIDInference* method), 194  
 Potential (class in *pyAgrum*), 48

`predict()` (*pyAgrum.skbn.BNClassifier method*), 252  
`predict_proba()` (*pyAgrum.skbn.BNClassifier method*), 252  
`PRMexplorer` (*class in pyAgrum*), 227  
`product()` (*pyAgrum.Potential method*), 53  
`property()` (*pyAgrum.BayesNetFragment method*), 87  
`propertyWithDefault()` (*pyAgrum.BayesNetFragment method*), 87  
`protectToLatex()` (*pyAgrum.causal.ASTBinaryOp method*), 240  
`protectToLatex()` (*pyAgrum.causal.ASTdiv method*), 243  
`protectToLatex()` (*pyAgrum.causal.ASTjointProba method*), 245  
`protectToLatex()` (*pyAgrum.causal.ASTminus method*), 242  
`protectToLatex()` (*pyAgrum.causal.ASTmult method*), 243  
`protectToLatex()` (*pyAgrum.causal.ASTplus method*), 241  
`protectToLatex()` (*pyAgrum.causal.ASTposteriorProba method*), 246  
`protectToLatex()` (*pyAgrum.causal.ASTsum method*), 244  
`protectToLatex()` (*pyAgrum.causal.ASTtree method*), 239  
`pseudoCount()` (*pyAgrum.BN Learner method*), 177  
`putFirst()` (*pyAgrum.Potential method*), 53  
`pyAgrum.causal.notebook` module, 247  
`PyAgrumConfiguration` (*class in pyAgrum*), 293

## R

`random()` (*pyAgrum.Potential method*), 54  
`randomCPT()` (*pyAgrum.Potential method*), 54  
`randomDistribution()` (*in module pyAgrum*), 284  
`randomDistribution()` (*pyAgrum.Potential method*), 54  
`randomProba()` (*in module pyAgrum*), 284  
`RangeVariable` (*class in pyAgrum*), 36  
`rawPseudoCount()` (*pyAgrum.BN Learner method*), 178  
`recordWeight()` (*pyAgrum.BN Learner method*), 178  
`reducedGraph()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 194  
`reducedLIMID()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 194  
`remainingBurnIn()` (*pyAgrum.GibbsBNdistance method*), 76  
`remove()` (*pyAgrum.Potential method*), 54  
`rend()` (*pyAgrum.Instantiation method*), 45  
`reorder()` (*pyAgrum.Instantiation method*), 45  
`reorganize()` (*pyAgrum.Potential method*), 54  
`reset()` (*pyAgrum.PyAgrumConfiguration method*), 293

`reverseArc()` (*pyAgrum.BayesNet method*), 70  
`reversePartialOrder()` (*pyAgrum.ShaferShenoyLIMIDInference method*), 194  
`root` (*pyAgrum.causal.CausalFormula property*), 237  
`run_hooks()` (*pyAgrum.PyAgrumConfiguration method*), 293

## S

`save()` (*pyAgrum.PyAgrumConfiguration method*), 294  
`saveBIF()` (*pyAgrum.BayesNet method*), 70  
`saveBIFXML()` (*pyAgrum.BayesNet method*), 70  
`saveBIFXML()` (*pyAgrum.InfluenceDiagram method*), 190  
`saveBN()` (*in module pyAgrum*), 280  
`saveBNsMinMax()` (*pyAgrum.CredalNet method*), 201  
`saveDSL()` (*pyAgrum.BayesNet method*), 70  
`saveID()` (*in module pyAgrum*), 281  
`saveInference()` (*pyAgrum.CNLoopyPropagation method*), 208  
`saveMN()` (*in module pyAgrum*), 281  
`saveNET()` (*pyAgrum.BayesNet method*), 70  
`saveO3PRM()` (*pyAgrum.BayesNet method*), 71  
`saveUAI()` (*pyAgrum.BayesNet method*), 71  
`saveUAI()` (*pyAgrum.MarkovNet method*), 217  
`scale()` (*pyAgrum.Potential method*), 54  
`score()` (*pyAgrum.skbn.BNClassifier method*), 252  
`second()` (*pyAgrum.Arc method*), 3  
`second()` (*pyAgrum.Edge method*), 4  
`separator()` (*pyAgrum.CliqueGraph method*), 17  
`set()` (*pyAgrum.Potential method*), 54  
`set()` (*pyAgrum.PyAgrumConfiguration method*), 294  
`set_params()` (*pyAgrum.skbn.BNClassifier method*), 253  
`setAntiTopologicalVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 72  
`setAprioriWeight()` (*pyAgrum.BN Learner method*), 178  
`setBurnIn()` (*pyAgrum.GibbsBNdistance method*), 76  
`setBurnIn()` (*pyAgrum.GibbsSampling method*), 122  
`setBurnIn()` (*pyAgrum.LoopyGibbsSampling method*), 150  
`setClique()` (*pyAgrum.CliqueGraph method*), 18  
`setCPT()` (*pyAgrum.CredalNet method*), 201  
`setCPTs()` (*pyAgrum.CredalNet method*), 202  
`setDatabaseWeight()` (*pyAgrum.BN Learner method*), 178  
`setDescription()` (*pyAgrum.DiscreteVariable method*), 26  
`setDescription()` (*pyAgrum.DiscretizedVariable method*), 32  
`setDescription()` (*pyAgrum.IntegerVariable method*), 35  
`setDescription()` (*pyAgrum.LabelizedVariable method*), 29

- setDescription() (pyAgrum.RangeVariable method), 38  
 setDiscretizationParameters() (pyAgrum.skbn.BNDiscretizer method), 255  
 setDrawnAtRandom() (pyAgrum.GibbsBNDistance method), 76  
 setDrawnAtRandom() (pyAgrum.GibbsSampling method), 122  
 setDrawnAtRandom() (pyAgrum.LoopyGibbsSampling method), 150  
 setEpsilon() (pyAgrum.BN Learner method), 178  
 setEpsilon() (pyAgrum.CN LoopyPropagation method), 208  
 setEpsilon() (pyAgrum.CN MonteCarloSampling method), 204  
 setEpsilon() (pyAgrum.GibbsBNDistance method), 77  
 setEpsilon() (pyAgrum.GibbsSampling method), 122  
 setEpsilon() (pyAgrum.ImportanceSampling method), 143  
 setEpsilon() (pyAgrum.LoopyBeliefPropagation method), 115  
 setEpsilon() (pyAgrum.LoopyGibbsSampling method), 150  
 setEpsilon() (pyAgrum.LoopyImportanceSampling method), 171  
 setEpsilon() (pyAgrum.LoopyMonteCarloSampling method), 157  
 setEpsilon() (pyAgrum.LoopyWeightedSampling method), 164  
 setEpsilon() (pyAgrum.MonteCarloSampling method), 129  
 setEpsilon() (pyAgrum.WeightedSampling method), 136  
 setEvidence() (pyAgrum.GibbsSampling method), 122  
 setEvidence() (pyAgrum.ImportanceSampling method), 143  
 setEvidence() (pyAgrum.LazyPropagation method), 96  
 setEvidence() (pyAgrum.LoopyBeliefPropagation method), 115  
 setEvidence() (pyAgrum.LoopyGibbsSampling method), 150  
 setEvidence() (pyAgrum.LoopyImportanceSampling method), 171  
 setEvidence() (pyAgrum.LoopyMonteCarloSampling method), 157  
 setEvidence() (pyAgrum.LoopyWeightedSampling method), 164  
 setEvidence() (pyAgrum.MonteCarloSampling method), 129  
 setEvidence() (pyAgrum.ShaferShenoyInference method), 103  
 setEvidence() (pyAgrum.ShaferShenoyLIMIDInference method), 194  
 setEvidence() (pyAgrum.ShaferShenoyMNIInference method), 224  
 setEvidence() (pyAgrum.VariableElimination method), 109  
 setEvidence() (pyAgrum.WeightedSampling method), 136  
 setFirst() (pyAgrum.Instantiation method), 46  
 setFirstIn() (pyAgrum.Instantiation method), 46  
 setFirstNotVar() (pyAgrum.Instantiation method), 46  
 setFirstOut() (pyAgrum.Instantiation method), 46  
 setFirstVar() (pyAgrum.Instantiation method), 46  
 setInitialDAG() (pyAgrum.BN Learner method), 178  
 setLast() (pyAgrum.Instantiation method), 46  
 setLastIn() (pyAgrum.Instantiation method), 46  
 setLastNotVar() (pyAgrum.Instantiation method), 46  
 setLastOut() (pyAgrum.Instantiation method), 46  
 setLastVar() (pyAgrum.Instantiation method), 46  
 setMaxIndegree() (pyAgrum.BN Learner method), 178  
 setMaxIter() (pyAgrum.BN Learner method), 178  
 setMaxIter() (pyAgrum.CN LoopyPropagation method), 208  
 setMaxIter() (pyAgrum.CN MonteCarloSampling method), 204  
 setMaxIter() (pyAgrum.GibbsBNDistance method), 77  
 setMaxIter() (pyAgrum.GibbsSampling method), 123  
 setMaxIter() (pyAgrum.ImportanceSampling method), 143  
 setMaxIter() (pyAgrum.LoopyBeliefPropagation method), 115  
 setMaxIter() (pyAgrum.LoopyGibbsSampling method), 150  
 setMaxIter() (pyAgrum.LoopyImportanceSampling method), 171  
 setMaxIter() (pyAgrum.LoopyMonteCarloSampling method), 157  
 setMaxIter() (pyAgrum.LoopyWeightedSampling method), 164  
 setMaxIter() (pyAgrum.MonteCarloSampling method), 129  
 setMaxIter() (pyAgrum.WeightedSampling method), 136  
 setMaxTime() (pyAgrum.BN Learner method), 178  
 setMaxTime() (pyAgrum.CN LoopyPropagation method), 208  
 setMaxTime() (pyAgrum.CN MonteCarloSampling method), 205  
 setMaxTime() (pyAgrum.GibbsBNDistance method), 77  
 setMaxTime() (pyAgrum.GibbsSampling method), 123  
 setMaxTime() (pyAgrum.ImportanceSampling

- method*), 143
- `setMaxTime()` (*pyAgrum.LoopyBeliefPropagation method*), 116
- `setMaxTime()` (*pyAgrum.LoopyGibbsSampling method*), 150
- `setMaxTime()` (*pyAgrum.LoopyImportanceSampling method*), 171
- `setMaxTime()` (*pyAgrum.LoopyMonteCarloSampling method*), 157
- `setMaxTime()` (*pyAgrum.LoopyWeightedSampling method*), 164
- `setMaxTime()` (*pyAgrum.MonteCarloSampling method*), 130
- `setMaxTime()` (*pyAgrum.WeightedSampling method*), 136
- `setMaxVal()` (*pyAgrum.RangeVariable method*), 38
- `setMinEpsilonRate()` (*pyAgrum.BN Learner method*), 179
- `setMinEpsilonRate()` (*pyAgrum.CNLoopyPropagation method*), 208
- `setMinEpsilonRate()` (*pyAgrum.CNMonteCarloSampling method*), 205
- `setMinEpsilonRate()` (*pyAgrum.GibbsBNdistance method*), 77
- `setMinEpsilonRate()` (*pyAgrum.GibbsSampling method*), 123
- `setMinEpsilonRate()` (*pyAgrum.ImportanceSampling method*), 143
- `setMinEpsilonRate()` (*pyAgrum.LoopyBeliefPropagation method*), 116
- `setMinEpsilonRate()` (*pyAgrum.LoopyGibbsSampling method*), 150
- `setMinEpsilonRate()` (*pyAgrum.LoopyImportanceSampling method*), 171
- `setMinEpsilonRate()` (*pyAgrum.LoopyMonteCarloSampling method*), 157
- `setMinEpsilonRate()` (*pyAgrum.LoopyWeightedSampling method*), 164
- `setMinEpsilonRate()` (*pyAgrum.MonteCarloSampling method*), 130
- `setMinEpsilonRate()` (*pyAgrum.WeightedSampling method*), 136
- `setMinVal()` (*pyAgrum.RangeVariable method*), 38
- `setMutable()` (*pyAgrum.Instantiation method*), 47
- `setName()` (*pyAgrum.DiscreteVariable method*), 26
- `setName()` (*pyAgrum.DiscretizedVariable method*), 32
- `setName()` (*pyAgrum.IntegerVariable method*), 35
- `setName()` (*pyAgrum.LabelizedVariable method*), 29
- `setName()` (*pyAgrum.RangeVariable method*), 38
- `setNbrDrawnVar()` (*pyAgrum.GibbsBNdistance method*), 77
- `setNbrDrawnVar()` (*pyAgrum.GibbsSampling method*), 123
- `setNbrDrawnVar()` (*pyAgrum.LoopyGibbsSampling method*), 151
- `setNumberOfThreads()` (*in module pyAgrum*), 285
- `setPeriodSize()` (*pyAgrum.BN Learner method*), 179
- `setPeriodSize()` (*pyAgrum.CNLoopyPropagation method*), 208
- `setPeriodSize()` (*pyAgrum.CNMonteCarloSampling method*), 205
- `setPeriodSize()` (*pyAgrum.GibbsBNdistance method*), 77
- `setPeriodSize()` (*pyAgrum.GibbsSampling method*), 123
- `setPeriodSize()` (*pyAgrum.ImportanceSampling method*), 143
- `setPeriodSize()` (*pyAgrum.LoopyBeliefPropagation method*), 116
- `setPeriodSize()` (*pyAgrum.LoopyGibbsSampling method*), 151
- `setPeriodSize()` (*pyAgrum.LoopyImportanceSampling method*), 171
- `setPeriodSize()` (*pyAgrum.LoopyMonteCarloSampling method*), 157
- `setPeriodSize()` (*pyAgrum.LoopyWeightedSampling method*), 164
- `setPeriodSize()` (*pyAgrum.MonteCarloSampling method*), 130
- `setPeriodSize()` (*pyAgrum.WeightedSampling method*), 136
- `setPossibleSkeleton()` (*pyAgrum.BN Learner method*), 179
- `setProperty()` (*pyAgrum.BayesNetFragment method*), 88
- `setRandomVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 72
- `setRecordWeight()` (*pyAgrum.BN Learner method*), 179
- `setRepetitiveInd()` (*pyAgrum.CNLoopyPropagation method*), 208
- `setRepetitiveInd()` (*pyAgrum.CNMonteCarloSampling method*), 205
- `setSliceOrder()` (*pyAgrum.BN Learner method*), 179
- `setTargets()` (*pyAgrum.GibbsSampling method*), 123
- `setTargets()` (*pyAgrum.ImportanceSampling method*), 143
- `setTargets()` (*pyAgrum.LazyPropagation method*), 96
- `setTargets()` (*pyAgrum.LoopyBeliefPropagation method*), 116
- `setTargets()` (*pyAgrum.LoopyGibbsSampling method*), 151



- method*), 151
- `setTargets()` (*pyAgrum.LoopyImportanceSampling method*), 172
- `setTargets()` (*pyAgrum.LoopyMonteCarloSampling method*), 158
- `setTargets()` (*pyAgrum.LoopyWeightedSampling method*), 165
- `setTargets()` (*pyAgrum.MonteCarloSampling method*), 130
- `setTargets()` (*pyAgrum.ShaferShenoyInference method*), 103
- `setTargets()` (*pyAgrum.ShaferShenoyMNIInference method*), 224
- `setTargets()` (*pyAgrum.VariableElimination method*), 110
- `setTargets()` (*pyAgrum.WeightedSampling method*), 137
- `setTopologicalVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 73
- `setVals()` (*pyAgrum.Instantiation method*), 47
- `setVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 73
- `setVarOrderFromCSV()` (*pyAgrum.BNDatabaseGenerator method*), 73
- `setVerbosity()` (*pyAgrum.BN Learner method*), 179
- `setVerbosity()` (*pyAgrum.CNLoopyPropagation method*), 208
- `setVerbosity()` (*pyAgrum.CNMonteCarloSampling method*), 205
- `setVerbosity()` (*pyAgrum.GibbsBNdistance method*), 77
- `setVerbosity()` (*pyAgrum.GibbsSampling method*), 123
- `setVerbosity()` (*pyAgrum.ImportanceSampling method*), 144
- `setVerbosity()` (*pyAgrum.LoopyBeliefPropagation method*), 116
- `setVerbosity()` (*pyAgrum.LoopyGibbsSampling method*), 151
- `setVerbosity()` (*pyAgrum.LoopyImportanceSampling method*), 172
- `setVerbosity()` (*pyAgrum.LoopyMonteCarloSampling method*), 158
- `setVerbosity()` (*pyAgrum.LoopyWeightedSampling method*), 165
- `setVerbosity()` (*pyAgrum.MonteCarloSampling method*), 130
- `setVerbosity()` (*pyAgrum.WeightedSampling method*), 137
- `setVirtualLBPSize()` (*pyAgrum.LoopyGibbsSampling method*), 151
- `setVirtualLBPSize()` (*pyAgrum.LoopyImportanceSampling method*), 172
- `setVirtualLBPSize()` (*pyAgrum.LoopyMonteCarloSampling method*), 158
- `setVirtualLBPSize()` (*pyAgrum.LoopyWeightedSampling method*), 165
- `ShaferShenoyInference` (*class in pyAgrum*), 97
- `ShaferShenoyLIMIDInference` (*class in pyAgrum*), 192
- `ShaferShenoyMNIInference` (*class in pyAgrum*), 218
- `showBN()` (*in module pyAgrum.lib.notebook*), 258
- `showCausalImpact()` (*in module pyAgrum.causal.notebook*), 247
- `showCausalModel()` (*in module pyAgrum.causal.notebook*), 247
- `showCN()` (*in module pyAgrum.lib.notebook*), 260
- `showDot()` (*in module pyAgrum.lib.notebook*), 263
- `showGraph()` (*in module pyAgrum.lib.notebook*), 263
- `showInference()` (*in module pyAgrum.lib.notebook*), 260
- `showInfluenceDiagram()` (*in module pyAgrum.lib.notebook*), 259
- `showInformation()` (*in module pyAgrum.lib.explain*), 267
- `showJunctionTree()` (*in module pyAgrum.lib.notebook*), 261
- `showMN()` (*in module pyAgrum.lib.notebook*), 259
- `showPosterior()` (*in module pyAgrum.lib.notebook*), 262
- `showPotential()` (*in module pyAgrum.lib.notebook*), 263
- `showProba()` (*in module pyAgrum.lib.notebook*), 262
- `showROC_PR()` (*pyAgrum.skbn.BNClassifier method*), 253
- `sideBySide()` (*in module pyAgrum.lib.notebook*), 264
- `size()` (*pyAgrum.BayesNet method*), 71
- `size()` (*pyAgrum.BayesNetFragment method*), 88
- `size()` (*pyAgrum.CliqueGraph method*), 18
- `size()` (*pyAgrum.DAG method*), 10
- `size()` (*pyAgrum.DiGraph method*), 7
- `size()` (*pyAgrum.EssentialGraph method*), 80
- `size()` (*pyAgrum.InfluenceDiagram method*), 190
- `size()` (*pyAgrum.MarkovBlanket method*), 82
- `size()` (*pyAgrum.MarkovNet method*), 217
- `size()` (*pyAgrum.MixedGraph method*), 22
- `size()` (*pyAgrum.UndiGraph method*), 13
- `sizeArcs()` (*pyAgrum.BayesNet method*), 71
- `sizeArcs()` (*pyAgrum.BayesNetFragment method*), 88
- `sizeArcs()` (*pyAgrum.DAG method*), 10
- `sizeArcs()` (*pyAgrum.DiGraph method*), 7
- `sizeArcs()` (*pyAgrum.EssentialGraph method*), 80
- `sizeArcs()` (*pyAgrum.InfluenceDiagram method*), 191
- `sizeArcs()` (*pyAgrum.MarkovBlanket method*), 82
- `sizeArcs()` (*pyAgrum.MixedGraph method*), 23
- `sizeEdges()` (*pyAgrum.CliqueGraph method*), 18
- `sizeEdges()` (*pyAgrum.EssentialGraph method*), 80
- `sizeEdges()` (*pyAgrum.MarkovNet method*), 217

`sizeEdges()` (*pyAgrum.MixedGraph* method), 23  
`sizeEdges()` (*pyAgrum.UndiGraph* method), 13  
`SizeError`, 290  
`sizeNodes()` (*pyAgrum.EssentialGraph* method), 80  
`sizeNodes()` (*pyAgrum.MarkovBlanket* method), 82  
`skeleton()` (*pyAgrum.EssentialGraph* method), 80  
`smallestFactorFromNode()` (*pyAgrum.MarkovNet* method), 217  
`softEvidenceNodes()` (*pyAgrum.GibbsSampling* method), 123  
`softEvidenceNodes()` (*pyAgrum.ImportanceSampling* method), 144  
`softEvidenceNodes()` (*pyAgrum.LazyPropagation* method), 96  
`softEvidenceNodes()` (*pyAgrum.LoopyBeliefPropagation* method), 116  
`softEvidenceNodes()` (*pyAgrum.LoopyGibbsSampling* method), 151  
`softEvidenceNodes()` (*pyAgrum.LoopyImportanceSampling* method), 172  
`softEvidenceNodes()` (*pyAgrum.LoopyMonteCarloSampling* method), 158  
`softEvidenceNodes()` (*pyAgrum.LoopyWeightedSampling* method), 165  
`softEvidenceNodes()` (*pyAgrum.MonteCarloSampling* method), 130  
`softEvidenceNodes()` (*pyAgrum.ShaferShenoyInference* method), 103  
`softEvidenceNodes()` (*pyAgrum.ShaferShenoyLIMIDInference* method), 194  
`softEvidenceNodes()` (*pyAgrum.ShaferShenoyMNIInference* method), 224  
`softEvidenceNodes()` (*pyAgrum.VariableElimination* method), 110  
`softEvidenceNodes()` (*pyAgrum.WeightedSampling* method), 137  
`sq()` (*pyAgrum.Potential* method), 54  
`src_bn()` (*pyAgrum.CredalNet* method), 202  
`startOfPeriod()` (*pyAgrum.GibbsBNdistance* method), 77  
`state()` (*pyAgrum.BNlearner* method), 179  
`stateApproximationScheme()` (*pyAgrum.GibbsBNdistance* method), 77  
`stopApproximationScheme()` (*pyAgrum.GibbsBNdistance* method), 78  
`stype()` (*pyAgrum.DiscreteVariable* method), 26  
`stype()` (*pyAgrum.DiscretizedVariable* method), 32  
`stype()` (*pyAgrum.IntegerVariable* method), 35  
`stype()` (*pyAgrum.LabelizedVariable* method), 30  
`stype()` (*pyAgrum.RangeVariable* method), 38  
`sum()` (*pyAgrum.Potential* method), 54

`SyntaxError`, 291

## T

`tail()` (*pyAgrum.Arc* method), 4  
`targets()` (*pyAgrum.GibbsSampling* method), 123  
`targets()` (*pyAgrum.ImportanceSampling* method), 144  
`targets()` (*pyAgrum.LazyPropagation* method), 96  
`targets()` (*pyAgrum.LoopyBeliefPropagation* method), 116  
`targets()` (*pyAgrum.LoopyGibbsSampling* method), 151  
`targets()` (*pyAgrum.LoopyImportanceSampling* method), 172  
`targets()` (*pyAgrum.LoopyMonteCarloSampling* method), 158  
`targets()` (*pyAgrum.LoopyWeightedSampling* method), 165  
`targets()` (*pyAgrum.MonteCarloSampling* method), 130  
`targets()` (*pyAgrum.ShaferShenoyInference* method), 104  
`targets()` (*pyAgrum.ShaferShenoyMNIInference* method), 224  
`targets()` (*pyAgrum.VariableElimination* method), 110  
`targets()` (*pyAgrum.WeightedSampling* method), 137  
`term` (*pyAgrum.causal.ASTsum* property), 244  
`thisown` (*pyAgrum.ArgumentError* property), 290  
`thisown` (*pyAgrum.BayesNet* property), 71  
`thisown` (*pyAgrum.CNLoopyPropagation* property), 209  
`thisown` (*pyAgrum.CPTErrors* 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), 288  
`thisown` (*pyAgrum.FormatNotFound* property), 288  
`thisown` (*pyAgrum.GibbsSampling* property), 124  
`thisown` (*pyAgrum.GraphError* property), 288  
`thisown` (*pyAgrum.ImportanceSampling* property), 144  
`thisown` (*pyAgrum.InfluenceDiagram* property), 191  
`thisown` (*pyAgrum.InvalidArc* property), 288  
`thisown` (*pyAgrum.InvalidArgument* property), 288  
`thisown` (*pyAgrum.InvalidArgumentsNumber* property), 289  
`thisown` (*pyAgrum.InvalidDirectedCycle* property), 289  
`thisown` (*pyAgrum.InvalidEdge* property), 289  
`thisown` (*pyAgrum.InvalidNode* property), 289  
`thisown` (*pyAgrum.IOError* property), 288  
`thisown` (*pyAgrum.LazyPropagation* property), 97  
`thisown` (*pyAgrum.LoopyBeliefPropagation* property), 116  
`thisown` (*pyAgrum.LoopyGibbsSampling* property), 151

- `thisown` (`pyAgrum.LoopyImportanceSampling` property), 172
- `thisown` (`pyAgrum.LoopyMonteCarloSampling` property), 158
- `thisown` (`pyAgrum.LoopyWeightedSampling` property), 165
- `thisown` (`pyAgrum.MarkovNet` property), 217
- `thisown` (`pyAgrum.MonteCarloSampling` property), 130
- `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), 55
- `thisown` (`pyAgrum.ShaferShenoyInference` property), 104
- `thisown` (`pyAgrum.ShaferShenoyMNIInference` property), 224
- `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), 291
- `thisown` (`pyAgrum.VariableElimination` property), 110
- `thisown` (`pyAgrum.WeightedSampling` property), 137
- `tick()` (`pyAgrum.DiscretizedVariable` method), 32
- `ticks()` (`pyAgrum.DiscretizedVariable` method), 33
- `toarray()` (`pyAgrum.Potential` method), 55
- `toBN()` (`pyAgrum.BayesNetFragment` method), 88
- `toclipboard()` (`pyAgrum.Potential` method), 55
- `toCSV()` (`pyAgrum.BNDatabaseGenerator` method), 73
- `toDatabaseTable()` (`pyAgrum.BNDatabaseGenerator` method), 73
- `todict()` (`pyAgrum.Instantiation` method), 47
- `toDiscretizedVar()` (`pyAgrum.DiscreteVariable` method), 26
- `toDiscretizedVar()` (`pyAgrum.DiscretizedVariable` method), 33
- `toDiscretizedVar()` (`pyAgrum.IntegerVariable` method), 35
- `toDiscretizedVar()` (`pyAgrum.LabelizedVariable` method), 30
- `toDiscretizedVar()` (`pyAgrum.RangeVariable` method), 38
- `toDot()` (`pyAgrum.BayesNet` method), 71
- `toDot()` (`pyAgrum.BayesNetFragment` method), 88
- `toDot()` (`pyAgrum.causal.CausalModel` method), 236
- `toDot()` (`pyAgrum.CliqueGraph` method), 18
- `toDot()` (`pyAgrum.DAG` method), 11
- `toDot()` (`pyAgrum.DiGraph` method), 7
- `toDot()` (`pyAgrum.EssentialGraph` method), 81
- `toDot()` (`pyAgrum.InfluenceDiagram` method), 191
- `toDot()` (`pyAgrum.MarkovBlanket` method), 82
- `toDot()` (`pyAgrum.MarkovNet` method), 217
- `toDot()` (`pyAgrum.MixedGraph` method), 23
- `toDot()` (`pyAgrum.UndiGraph` method), 13
- `toDotAsFactorGraph()` (`pyAgrum.MarkovNet` method), 217
- `toDotWithNames()` (`pyAgrum.CliqueGraph` method), 18
- `toIntegerVar()` (`pyAgrum.DiscreteVariable` method), 26
- `toIntegerVar()` (`pyAgrum.DiscretizedVariable` method), 33
- `toIntegerVar()` (`pyAgrum.IntegerVariable` method), 35
- `toIntegerVar()` (`pyAgrum.LabelizedVariable` method), 30
- `toIntegerVar()` (`pyAgrum.RangeVariable` method), 39
- `toLabelizedVar()` (`pyAgrum.DiscreteVariable` method), 26
- `toLabelizedVar()` (`pyAgrum.DiscretizedVariable` method), 33
- `toLabelizedVar()` (`pyAgrum.IntegerVariable` method), 35
- `toLabelizedVar()` (`pyAgrum.LabelizedVariable` method), 30
- `toLabelizedVar()` (`pyAgrum.RangeVariable` method), 39
- `toLatex()` (`pyAgrum.causal.ASTBinaryOp` method), 240
- `toLatex()` (`pyAgrum.causal.ASTdiv` method), 243
- `toLatex()` (`pyAgrum.causal.ASTjointProba` method), 245
- `toLatex()` (`pyAgrum.causal.ASTminus` method), 242
- `toLatex()` (`pyAgrum.causal.ASTMult` method), 244
- `toLatex()` (`pyAgrum.causal.ASTplus` method), 241
- `toLatex()` (`pyAgrum.causal.ASTposteriorProba` method), 246
- `toLatex()` (`pyAgrum.causal.ASTsum` method), 244
- `toLatex()` (`pyAgrum.causal.ASTtree` method), 239
- `toLatex()` (`pyAgrum.causal.CausalFormula` method), 237
- `tolatex()` (`pyAgrum.Potential` method), 55
- `tolist()` (`pyAgrum.Potential` method), 55
- `topandas()` (`pyAgrum.Potential` method), 55
- `topologicalOrder()` (`pyAgrum.BayesNet` method), 71
- `topologicalOrder()` (`pyAgrum.BayesNetFragment` method), 88
- `topologicalOrder()` (`pyAgrum.DAG` method), 11
- `topologicalOrder()` (`pyAgrum.DiGraph` method), 7
- `topologicalOrder()` (`pyAgrum.InfluenceDiagram` method), 191
- `topologicalOrder()` (`pyAgrum.MixedGraph` method), 23



method), 23  
 toRangeVar() (pyAgrum.DiscreteVariable method), 26  
 toRangeVar() (pyAgrum.DiscretizedVariable method), 33  
 toRangeVar() (pyAgrum.IntegerVariable method), 35  
 toRangeVar() (pyAgrum.LabelizedVariable method), 30  
 toRangeVar() (pyAgrum.RangeVariable method), 39  
 toStringWithDescription() (pyAgrum.DiscreteVariable method), 27  
 toStringWithDescription() (pyAgrum.DiscretizedVariable method), 33  
 toStringWithDescription() (pyAgrum.IntegerVariable method), 36  
 toStringWithDescription() (pyAgrum.LabelizedVariable method), 30  
 toStringWithDescription() (pyAgrum.RangeVariable method), 39  
 translate() (pyAgrum.Potential method), 55  
 type (pyAgrum.causal.ASTBinaryOp property), 240  
 type (pyAgrum.causal.ASTdiv property), 243  
 type (pyAgrum.causal.ASTjointProba property), 245  
 type (pyAgrum.causal.ASTminus property), 242  
 type (pyAgrum.causal.ASTMult property), 244  
 type (pyAgrum.causal.ASTplus property), 241  
 type (pyAgrum.causal.ASTposteriorProba property), 246  
 type (pyAgrum.causal.ASTsum property), 245  
 type (pyAgrum.causal.ASTtree property), 239  
 types() (pyAgrum.PRMexplorer method), 231

## U

UndefinedElement, 291  
 UndefinedIteratorKey, 291  
 UndefinedIteratorValue, 291  
 UndiGraph (class in pyAgrum), 11  
 UnidentifiableException (class in pyAgrum.causal), 247  
 uninstallCPT() (pyAgrum.BayesNetFragment method), 88  
 uninstallNode() (pyAgrum.BayesNetFragment method), 88  
 UnknownLabelInDatabase, 291  
 unsetEnd() (pyAgrum.Instantiation method), 47  
 unsetOverflow() (pyAgrum.Instantiation method), 47  
 updateApproximationScheme() (pyAgrum.GibbsBNDistance method), 78  
 updateEvidence() (pyAgrum.GibbsSampling method), 124  
 updateEvidence() (pyAgrum.ImportanceSampling method), 144  
 updateEvidence() (pyAgrum.LazyPropagation method), 97  
 updateEvidence() (pyAgrum.LoopyBeliefPropagation method), 116

updateEvidence() (pyAgrum.LoopyGibbsSampling method), 151  
 updateEvidence() (pyAgrum.LoopyImportanceSampling method), 172  
 updateEvidence() (pyAgrum.LoopyMonteCarloSampling method), 158  
 updateEvidence() (pyAgrum.LoopyWeightedSampling method), 165  
 updateEvidence() (pyAgrum.MonteCarloSampling method), 130  
 updateEvidence() (pyAgrum.ShaferShenoyInference method), 104  
 updateEvidence() (pyAgrum.ShaferShenoyLIMIDInference method), 194  
 updateEvidence() (pyAgrum.ShaferShenoyMNIInference method), 224  
 updateEvidence() (pyAgrum.VariableElimination method), 110  
 updateEvidence() (pyAgrum.WeightedSampling method), 137  
 use3off2() (pyAgrum.BN Learner method), 179  
 useAprioriBDeu() (pyAgrum.BN Learner method), 179  
 useAprioriDirichlet() (pyAgrum.BN Learner method), 179  
 useAprioriSmoothing() (pyAgrum.BN Learner method), 179  
 useEM() (pyAgrum.BN Learner method), 180  
 useGreedyHillClimbing() (pyAgrum.BN Learner method), 180  
 useK2() (pyAgrum.BN Learner method), 180  
 useLocalSearchWithTabuList() (pyAgrum.BN Learner method), 180  
 useMDLCorrection() (pyAgrum.BN Learner method), 180  
 useMIIC() (pyAgrum.BN Learner method), 180  
 useNMLCorrection() (pyAgrum.BN Learner method), 180  
 useNoApriori() (pyAgrum.BN Learner method), 180  
 useNoCorrection() (pyAgrum.BN Learner method), 180  
 useScoreAIC() (pyAgrum.BN Learner method), 180  
 useScoreBD() (pyAgrum.BN Learner method), 180  
 useScoreBDeu() (pyAgrum.BN Learner method), 180  
 useScoreBIC() (pyAgrum.BN Learner method), 181  
 useScoreK2() (pyAgrum.BN Learner method), 181  
 useScoreLog2Likelihood() (pyAgrum.BN Learner method), 181  
 utility() (pyAgrum.InfluenceDiagram method), 191  
 utilityNodeSize() (pyAgrum.InfluenceDiagram method), 191



## V

`val()` (*pyAgrum.Instantiation method*), 47  
`var_dims` (*pyAgrum.Potential property*), 55  
`var_names` (*pyAgrum.Potential property*), 55  
`variable()` (*pyAgrum.BayesNet method*), 71  
`variable()` (*pyAgrum.BayesNetFragment method*), 88  
`variable()` (*pyAgrum.InfluenceDiagram method*), 191  
`variable()` (*pyAgrum.Instantiation method*), 47  
`variable()` (*pyAgrum.MarkovNet method*), 217  
`variable()` (*pyAgrum.Potential method*), 55  
`VariableElimination` (*class in pyAgrum*), 104  
`variableFromName()` (*pyAgrum.BayesNet method*), 71  
`variableFromName()` (*pyAgrum.BayesNetFragment method*), 89  
`variableFromName()` (*pyAgrum.InfluenceDiagram method*), 191  
`variableFromName()` (*pyAgrum.MarkovNet method*), 217  
`variableNodeMap()` (*pyAgrum.BayesNet method*), 72  
`variableNodeMap()` (*pyAgrum.BayesNetFragment method*), 89  
`variableNodeMap()` (*pyAgrum.InfluenceDiagram method*), 191  
`variableNodeMap()` (*pyAgrum.MarkovNet method*), 217  
`variablesSequence()` (*pyAgrum.Instantiation method*), 47  
`variablesSequence()` (*pyAgrum.Potential method*), 55  
`varNames` (*pyAgrum.causal.ASTjointProba property*), 245  
`varOrder()` (*pyAgrum.BNDatabaseGenerator method*), 73  
`varOrderNames()` (*pyAgrum.BNDatabaseGenerator method*), 73  
`vars` (*pyAgrum.causal.ASTposteriorProba property*), 246  
`varType()` (*pyAgrum.DiscreteVariable method*), 27  
`varType()` (*pyAgrum.DiscretizedVariable method*), 33  
`varType()` (*pyAgrum.IntegerVariable method*), 36  
`varType()` (*pyAgrum.LabelizedVariable method*), 30  
`varType()` (*pyAgrum.RangeVariable method*), 39  
`verbosity()` (*pyAgrum.BN Learner method*), 181  
`verbosity()` (*pyAgrum.CN LoopyPropagation method*), 209  
`verbosity()` (*pyAgrum.CN MonteCarloSampling method*), 205  
`verbosity()` (*pyAgrum.GibbsBN distance method*), 78  
`verbosity()` (*pyAgrum.GibbsSampling method*), 124  
`verbosity()` (*pyAgrum.ImportanceSampling method*), 144  
`verbosity()` (*pyAgrum.LoopyBeliefPropagation method*), 117  
`verbosity()` (*pyAgrum.LoopyGibbsSampling method*), 152

`verbosity()` (*pyAgrum.LoopyImportanceSampling method*), 172  
`verbosity()` (*pyAgrum.LoopyMonteCarloSampling method*), 158  
`verbosity()` (*pyAgrum.LoopyWeightedSampling method*), 165  
`verbosity()` (*pyAgrum.MonteCarloSampling method*), 131  
`verbosity()` (*pyAgrum.WeightedSampling method*), 137  
`VI()` (*pyAgrum.LazyPropagation method*), 90  
`VI()` (*pyAgrum.ShaferShenoyInference method*), 98  
`VI()` (*pyAgrum.ShaferShenoyMNIInference method*), 218

## W

`WeightedSampling` (*class in pyAgrum*), 131  
`what()` (*pyAgrum.ArgumentError method*), 290  
`what()` (*pyAgrum.CPTErrror method*), 292  
`what()` (*pyAgrum.DatabaseError method*), 292  
`what()` (*pyAgrum.DuplicateElement method*), 287  
`what()` (*pyAgrum.DuplicateLabel method*), 288  
`what()` (*pyAgrum.FatalError method*), 288  
`what()` (*pyAgrum.FormatNotFound method*), 288  
`what()` (*pyAgrum.GraphError method*), 288  
`what()` (*pyAgrum.GumException method*), 287  
`what()` (*pyAgrum.InvalidArc method*), 288  
`what()` (*pyAgrum.InvalidArgument method*), 288  
`what()` (*pyAgrum.InvalidArgumentsNumber method*), 289  
`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*), 290  
`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*), 291  
`whenArcAdded()` (*pyAgrum.BayesNetFragment method*), 89  
`whenArcDeleted()` (*pyAgrum.BayesNetFragment method*), 89  
`whenNodeAdded()` (*pyAgrum.BayesNetFragment method*), 89  
`whenNodeDeleted()` (*pyAgrum.BayesNetFragment method*), 89

`with_traceback()` (*pyAgrum.GumException*  
*method*), [287](#)

## X

`XYfromCSV()` (*pyAgrum.skbn.BNClassifier* *method*),  
[251](#)