pyAgrum Documentation

Release 0.22.2

Pierre-Henri Wuillemin (Sphinx)

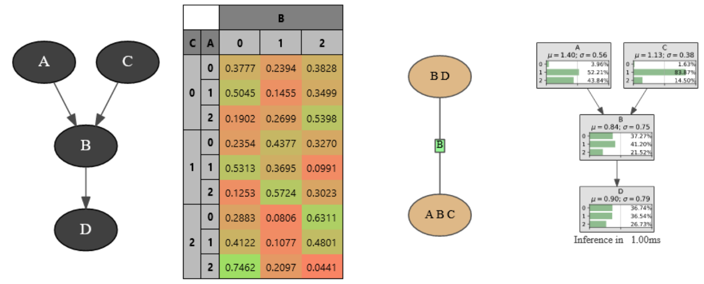
September 21, 2021

FUNDAMENTAL COMPONENTS

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

(http://agrum.org)

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



		B		
C	A	0	1	2
0	0	0.3777	0.2394	0.3828
	1	0.5045	0.1455	0.3499
	2	0.1902	0.2699	0.5398
1	0	0.2354	0.4377	0.3270
	1	0.5313	0.3695	0.0991
	2	0.1253	0.5724	0.3023
2	0	0.2883	0.0806	0.6311
	1	0.4122	0.1077	0.4801
	2	0.7462	0.2097	0.0441

a BN | a CPT : $P(B|A,C)$ | a Junction Tree | an inference

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

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

pyAgrum contains :

- a comprehensive API documentation (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).

CHAPTER

ONE

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 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

moralGraph()

Return type *pyAgrum.UndiGraph* (page 11)

moralizedAncestralGraph(*nodes*)

Parameters **nodes** (List[int]) –

Return type *pyAgrum.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 bewteen n1 and n2.

Parameters

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

Returns True if the arc exists

Return type bool

`existsNode(id)`
Check if a node with a certain id exists in the graph.

Parameters **id** (*int*) – the checked id

Returns True if the node exists

Return type bool

`hasUndirectedCycle()`
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

neighbours(*id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes()

Returns the set of ids

Return type set

nodes2ConnectedComponent()

Return type Dict[int, int]

partialUndiGraph(*nodes*)

Parameters

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

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 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** (pyAgrum.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 bewteen n1 and n2.

Parameters

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

Parameters **id** (int) –

Return type List[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 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 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

`existsEdge`(*n1*, *n2*)
Check if an edge exists bewteen n1 and n2.

Parameters

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

`hasUndirectedCycle`()
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

`mixedOrientedPath`(*node1*, *node2*)

Parameters

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

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

Return type List

`mixedUnorientedPath`(*node1*, *node2*)

Parameters

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

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

Return type List

`neighbours(id)`

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

Returns The set of edges adjacent to the given node

Return type Set

`nodes()`

Returns the set of ids

Return type set

`nodes2ConnectedComponent()`

Return type `Dict[int, int]`

`parents(id)`

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

Returns the set of the parents ids.

Return type Set

`partialUndiGraph(nodes)`

Parameters

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

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 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`) –

CHAPTER

TWO

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.OutOfBound` – 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 *pyAgrum.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: LabelizedVariable, 2: RangeVariable

Return type int

2.2 Concrete classes for Random Discrete Variables

2.2.1 LabelizedVariable

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

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

LabelizedVariable(aName, aDesc='', nbrLabel=2) -> LabelizedVariable

Parameters:

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

LabelizedVariable(aLDRV) -> LabelizedVariable

Parameters:

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

Examples

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

(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 labelized variable

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 27)

Raises **gum.DuplicateElement** – 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 labelized variable

Raises

- **pyAgrum.DuplicatedElement** – If the variable already contains the new label
- ***pyAgrum.OutOfBounds*** (page 291) – 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.OutOfBound` – 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 *pyAgrum.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: 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 **gum.DefaultInLabel** – 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.OutOfBound` – 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 *pyAgrum.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: LabelizedVariable, 2: RangeVariable

Return type int

2.2.3 IntegerVariable

class `pyAgrum.IntegerVariable(*args)`

`addValue(value)`

Parameters **value** (int) –

Return type *pyAgrum.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.OutOfBound** – 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 *pyAgrum.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: LabelizedVariable, 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
```

(continues on next page)

(continued from previous page)

```
>>> 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.OutOfBound` – 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 *pyAgrum.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: LabelizedVariable, 2: RangeVariable

Return type int

CHAPTER

THREE

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)).

```
>>> 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
```

(continues on next page)

(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.
- **OutOfBound** – 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.

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

Parameters **dict** (`object`) –

Return type `None`

`hamming()`

Returns the hamming distance of this instantiation.

Return type int

`inOverflow()`

Returns True if the current value of the tuple is correct

Return type bool

`inc()`

Operator ++.

Return type `None`

`incIn(i)`
Operator ++ for the variables in i.

Parameters **i** (`pyAgrum.Instantiation` (page 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 the sequence of DiscreteVariable of this instantiation.

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 289) – 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 289) – 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 dictionnary 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 potentia

Return type *pyAgrum.Potential* (page 48)

Raises **pyAgrum.SizeError** (page 291) – 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 **gum.InvalidArgument** – 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 *pyAgrum.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 *pyAgrum.Potential* (page 48)

`loopIn()`

Generator to iterate inside a Potential.

Yield an gum.Instantiation that iterates over all the possible values for the gum.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 289) – 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 289) – 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 289) – 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 *pyAgrum.Potential* (page 48)

`new_log2()`

Return type *pyAgrum.Potential* (page 48)

`new_sq()`

Return type *pyAgrum.Potential* (page 48)

`noising(alpha)`

Parameters **alpha** (`float`) –

Return type *pyAgrum.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.

Returns

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 289) – If the var is not in the potential

`random()`

Return type *pyAgrum.Potential* (page 48)

`randomCPT()`

Return type *pyAgrum.Potential* (page 48)

`randomDistribution()`

Return type *pyAgrum.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)

> **Warning:** IndexError raised if the var is not in the potential

Parameters **var** (`pyAgrum.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

Returns

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 *pyAgrum.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

Returns

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

> **Warning:** listed in the reverse order of the enumeration order of the variables.

variable(**args*)

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

Returns

Return type the varible at the ith index

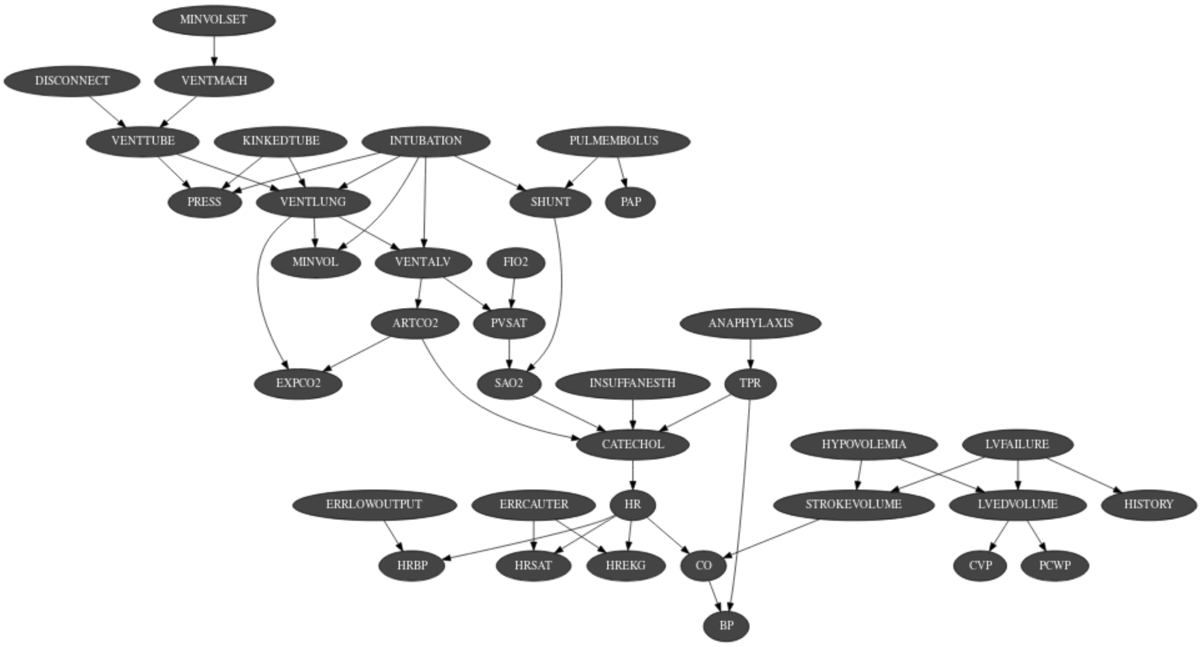
Raises *pyAgrum.NotFound* (page 290) – If i does not reference a variable in this multidimensional matrix.

`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.

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

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)

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 288) – 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** (pyAgrum.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** (pyAgrum.DiscreteVariable (page 25)) –

Returns the id of the added variable.

Return type int

Raises *pyAgrum.SizeError* (page 291) – 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** (pyAgrum.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** (pyAgrum.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** (pyAgrum.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** (pyAgrum.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** (pyAgrum.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** (pyAgrum.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** (`pyAgrum.DiscreteVariable` (page 25)) –

Returns the id of the added variable.

Return type int

Raises ***pyAgrum.SizeError*** (page 291) – If variable.domainSize()>2

addSUM(*var*)
Others aggregators

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 25)) – the variable to be added
- **var** (pyAgrum.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 lisf 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 288) – 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 *pyAgrum.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 7)

`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.

Returns

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
- **other** (*pyAgrum.DAGmodel*) –

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*)

Parameters **names** (*Vector_string*) –

Return type pyAgrum.YetUnWrapped

`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 DiscretizedVariable but also RangeVariable and LabelizedVariable.

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 *pyAgrum.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 exsit or if its reversal would induce a directed cycle.

Return type `None`

`saveBIF`(*name*)

Save the BayesNet in a BIF file.

Parameters **name** (*str*) – the file's name

Return type `None`

`saveBIFXML`(*name*)

Save the BayesNet in a BIFXML file.

Parameters **name** (*str*) – the file's name

Return type `None`

`saveDSL`(*name*)

Save the BayesNet in a DSL file.

Parameters **name** (*str*) – the file's name

Return type None

saveNET(*name*)
Save the BayesNet in a NET file.

Parameters **name** (*str*) – the file's name

Return type None

saveO3PRM(*name*)
Save the BayesNet in an O3PRM file.

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

Parameters **name** (*str*) – the file's name

Return type None

saveUAI(*name*)
Save the BayesNet in an UAI file.

Parameters **name** (*str*) – the file's name

Return type None

size()

Returns the number of nodes in the graph

Return type int

sizeArcs()

Returns the number of arcs in the graph

Return type int

property thisown
The membership flag

toDot()

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises *pyAgrum.InvalidDirectedCycle* (page 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)

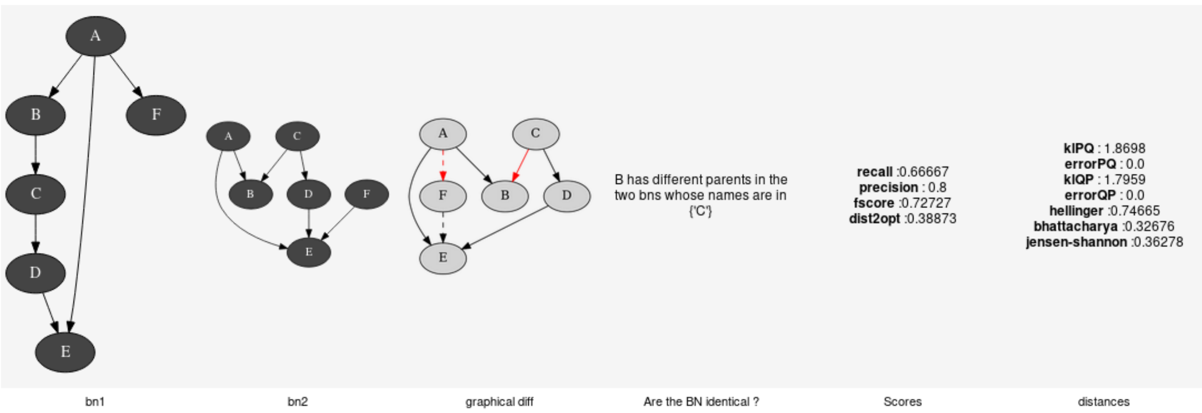
Raises **pyAgrum.IndexError** – If the graph does not contain the variable

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 easely 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()

Return type pyAgrum.YetUnWrapped

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)`

Parameters **useLabels** (`bool`) –

Return type pyAgrum.YetUnWrapped

`varOrder()`

Return type pyAgrum.YetUnWrapped

`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)`

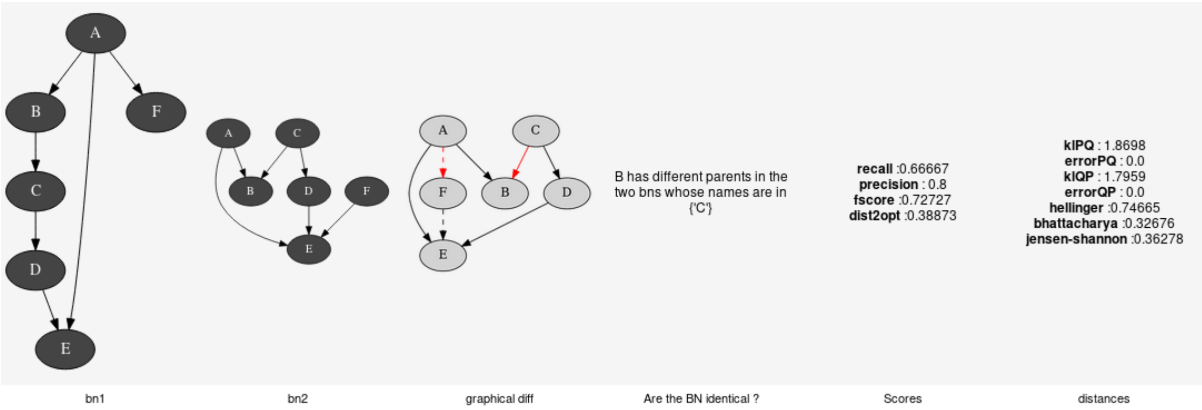
Class representing exacte computation of divergence and distance between BNs

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 dictionnary 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 dictionnary 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 291) – 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 291) – If eps<0

Return type None

setMaxIter(*max*)

Parameters **max** (*int*) – the maximum number of iteration

Raises *pyAgrum.OutOfBounds* (page 291) – If max <= 1

Return type None

setMaxTime(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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) –

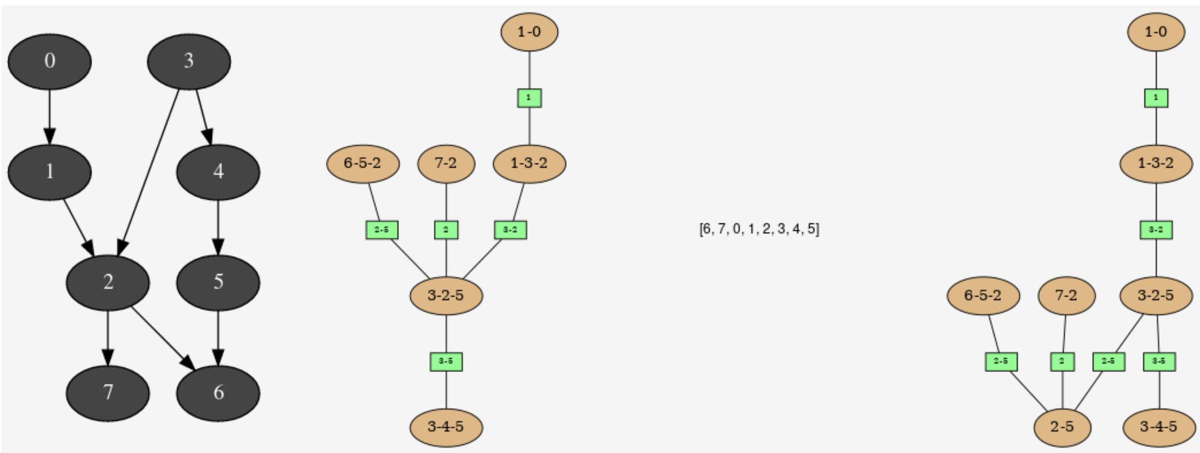
Return type None

`verbosity()`

Returns True if the verbosity is enabled

Return type bool

4.2.3 Explanation and analysis



A Bayesien network   a junction tree for this BN   its elimination order   an (optimized) binary join tree

This tools aimed to provide some different views on the Bayesian network in order to explore its qualitative and/or quantitave 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 heurisitcs 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 7)) – 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 13)

`eliminationOrder(*args)`

Computes the elimination for its parameters. If the first parameter is a graph, the heurisitcs 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 7)) – 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 13)

`junctionTree(*args)`

Computes the junction tree for its parameters. If the first parameter is a graph, the heurisitcs 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 7)) – 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 13)

`class pyAgrum.EssentialGraph(*args)`

`arcs()`

Returns The lisf 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 *pyAgrum.UndiGraph* (page 11)

`toDot()`

Returns a friendly display of the graph in DOT format

Return type str

`class pyAgrum.MarkovBlanket(*args)`

`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 7)

`hasSameStructure(other)`

Parameters

- **pyAgrum.DAGmodel** – a direct acyclic model
- **other** (*pyAgrum.DAGmodel*) –

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 lisf 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 consistant for this variable. This function checks the consistency for a variable of for all.

Parameters **n** (*int, str (optional)*) – the id or the name of the variable. If no argument, the function checks all the variables.

Returns True if the variable(s) is consistant.

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 *pyAgrum.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 7)

`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
- **other** (*pyAgrum.DAGmodel*) –

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)`

Parameters **names** (*Vector_string*) –

Return type pyAgrum.YetUnWrapped

`installAscendants(*args)`
Add the variable and all its ascendants in the fragment. No inconsistant 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 consistant, or the parents are not in the fragment and the node is not consistant.

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 *pyAgrum.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 *pyAgrum.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 inconsistant.

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 inconsistant.

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 probabilistc 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 thant 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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 instanciation 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 evidene 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 292) – 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 292) – If node Id is not in the Bayesian network

`joinTree()`

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 13)

`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 bu 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 292) – 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 13)

`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 292) – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s

- **gum.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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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 instanciation 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 evidene 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 292) – 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 292) – If node Id is not in the Bayesian network

`joinTree`()

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 13)

`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 bu 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 292) – 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 13)

`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 292) – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.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 `gum.UndefinedElement` – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.UndefinedElement` – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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(targets, evs)`
Create a pyAgrum.Potential for P(joint targets|evs) (for all instanciation 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(target|evs)

Return type *pyAgrum.Potential* (page 48)

Raises `pyAgrum.Exception` – If some evidene 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 292) – 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 292) – 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 bu 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 292) – 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 13)

Parameters **id** (`int`) –

`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 292) – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.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 `gum.UndefinedElement` – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node

- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node

- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 instanciation 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)

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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon`(*eps*)

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

Raises `pyAgrum.OutOfBounds` (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime`(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises `pyAgrum.OutOfBounds` (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

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 292) – 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 291) – 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 292) – 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 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

setMaxTime(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises **pyAgrum.OutOfBounds** (page 291) – 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 291) – 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 `gum.UndefinedElement` – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.UndefinedElement` – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

`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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon`(*eps*)

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

Raises `pyAgrum.OutOfBounds` (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime`(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises `pyAgrum.OutOfBounds` (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

setEpsilon(*eps*)

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

Raises *pyAgrum.OutOfBounds* (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

setMaxTime(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.UndefinedElement` – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon(eps)`

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

Raises `pyAgrum.OutOfBounds` (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node

- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime`(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node

- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

`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 292) – 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 291) – 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 292) – 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 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

setMaxTime(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

`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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon`(*eps*)

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

Raises *pyAgrum.OutOfBounds* (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime`(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 `gum.UndefinedElement` – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.UndefinedElement` – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

`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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon(eps)`

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

Raises `pyAgrum.OutOfBounds` (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime(timeout)`

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises `pyAgrum.OutOfBounds` (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 292) – 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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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)

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 292) – 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 291) – 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 292) – If an element of nodes is not in targets

`setEpsilon(eps)`

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

Raises *pyAgrum.OutOfBounds* (page 291) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 291) – If max <= 1

Return type None

`setMaxTime(timeout)`

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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 conditioanl likelihood on the database, etc.).

`class pyAgrum.BNLearner(filename, inducedTypes=True)` → BNLearner

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 7)

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 7)) –
- **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 292) – 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 nimber 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 291) – 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

Returns

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 BNLearner 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 291) – If eps<0

Return type None

setInitialDAG(*g*)

Parameters

- **dag** (pyAgrum.DAG (page 7)) – an initial DAG structure
- **g** (*DAG* (page 7)) –

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 291) – If max <= 1

Return type None

setMaxTime(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)

- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – If p<1

Return type None

setPossibleSkeleton(*skeleton*)

Parameters **skeleton** (pyAgrum.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 les 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()`

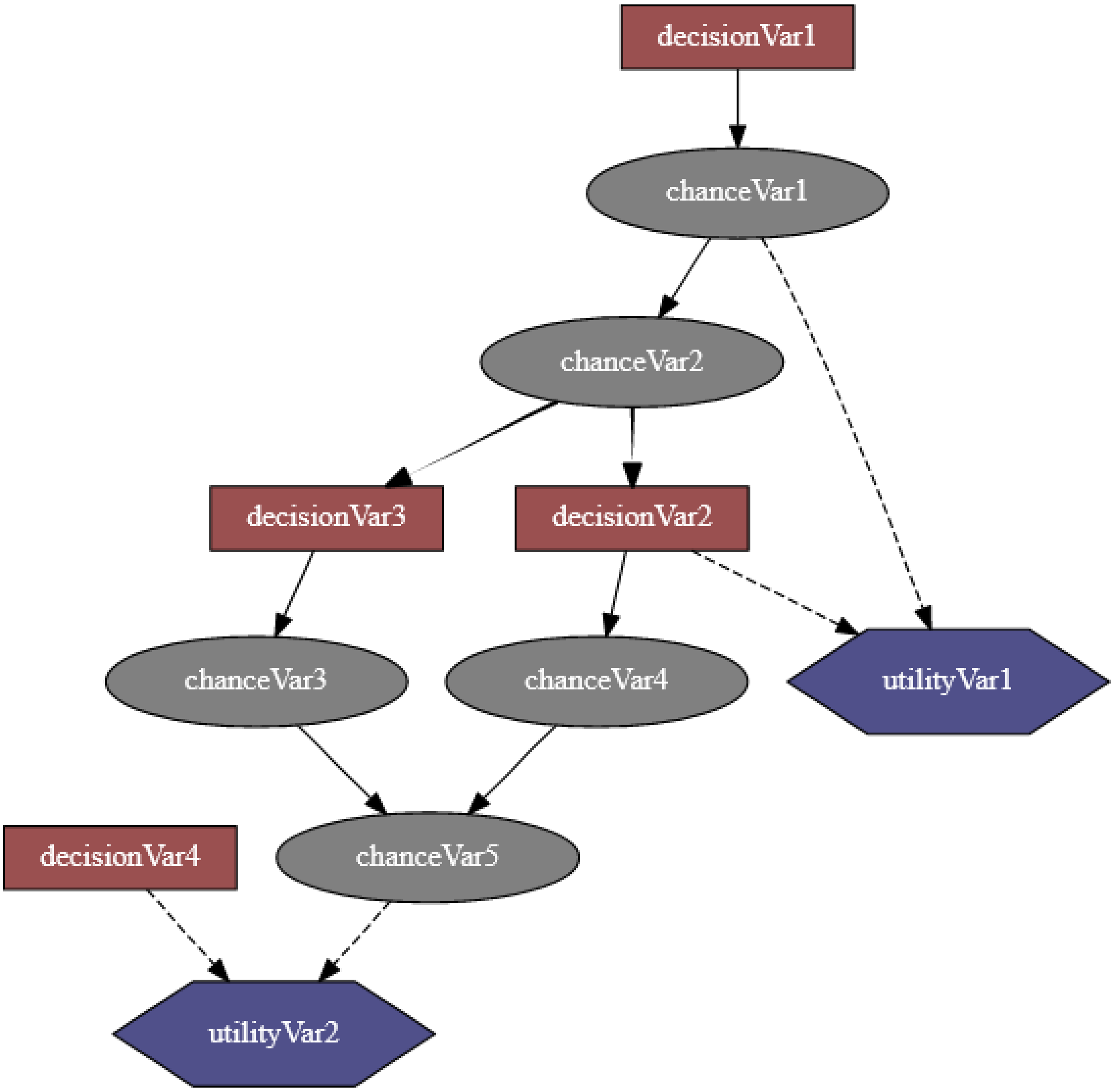
Returns True if the verbosity is enabled

Return type bool

CHAPTER

FIVE

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)).

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.ht

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 289) – 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 288) – 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 *pyAgrum.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 7)

`decisionNodeSize()`

Returns the number of decision nodes

Return type int

`decisionOrder()`

Return type pyAgrum.YetUnWrapped

`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/utiliy 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 7)

hasSameStructure(*other*)

Parameters

- **pyAgrum.DAGmodel** – a direct acyclic model
- **other** (*pyAgrum.DAGmodel*) –

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*)

Parameters **names** (*Vector_string*) –

Return type pyAgrum.YetUnWrapped

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 *pyAgrum.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

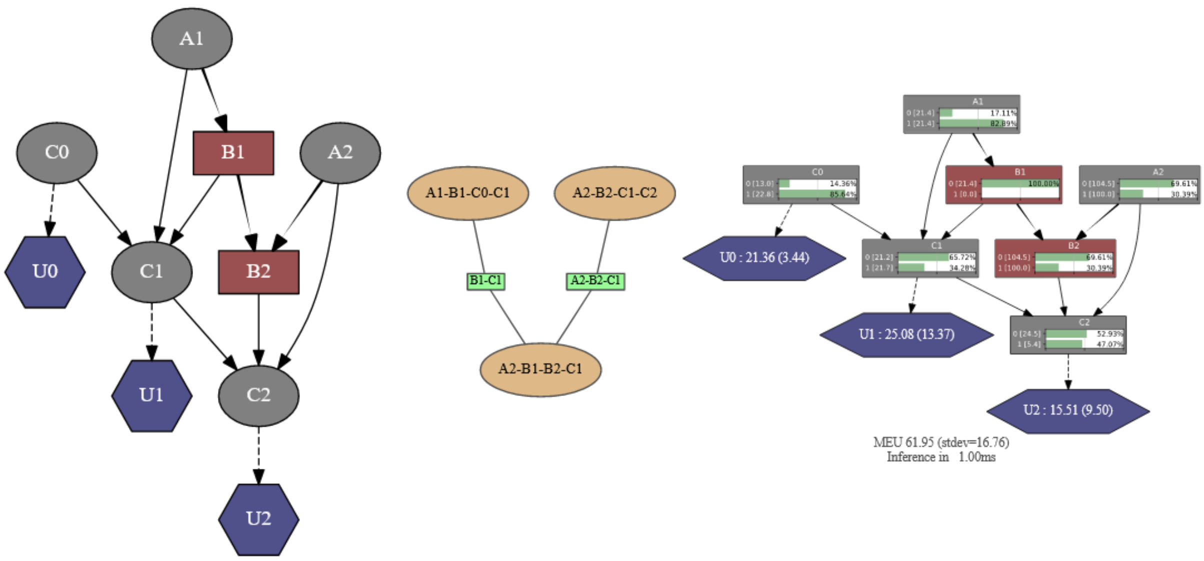
Return type *pyAgrum.DiscreteVariable* (page 25)

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

`variableNodeMap()`

Return type pyAgrum.VariableNodeMap

5.2 Inference



An influence diagram | Its reduced Junction Tree | The inference

	B1	
A1	0	1
0	1.0000	0.0000
1	1.0000	0.0000

Optimal decision for B_1

	B2	
A2	0	1
0	1.0000	0.0000
1	0.0000	1.0000

Optimal decision for B_2

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()

Return type pyAgrum.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** (page 289) – If node given in parmaeter is not a decision node

Return type *pyAgrum.Potential* (page 48)

posterior(**args*)

Return type *pyAgrum.Potential* (page 48)

posteriorUtility(**args*)

Return type *pyAgrum.Potential* (page 48)

reducedGraph()

Return type *pyAgrum.DAG* (page 7)

`reducedLIMID()`

Return type *pyAgrum.InfluenceDiagram* (page 184)

`reversePartialOrder()`

Return type pyAgrum.YetUnWrapped

`setEvidence(`*evidces*`)`

Erase all the evidences and apply addEvidence(key,value) for every pairs in evidces.

Parameters **evidces** (*dict*) – a dict of evidences

Raises

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.UndefinedElement** – If one node does not belong to the Bayesian network

CHAPTER

SIX

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)).

Tutorial

- Tutorial on Credal Networks (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 orignal credal sets and may induce huge imprecision.

> **Warning:** Enlarge the orignal 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

Warning: You need to call intervalToCredal when done filling all constraints.

Warning: DOES change the BayesNet (s) associated to this credal net !

Return type `None`

`fillConstraints`(*id*, *lower*, *upper*)
Set the interval constraints of the credal sets of a given node (all instantiations)

Parameters

- **id** (*int*) – The id of the node
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

Warning: You need to call intervalToCredal when done filling all constraints.

Warning: DOES change the BayesNet (s) associated to this credal net !

Return type `None`

`get_binaryCPT_max`()

Warning: Experimental function - Return type to be wrapped

Returns a constant reference to the upper probabilities of each node X over the 'True' modality

Return type tbw

`get_binaryCPT_min`()

Warning: Experimental function - Return type to be wrapped

Returns a constant reference to the lower probabilities of each node X over the 'True' modality

Return type tbw

`hasComputedBinaryCPTMinMax`()

Return type `bool`

`idmLearning`(*s=0*, *keepZeroes=False*)
Learns parameters from a BayesNet storing counts of events.

Use this method when using a single BayesNet storing counts of events. IDM model if s > 0, standard point probability if s = 0 (default value if none precised).

Parameters

- **s** (*int*) – the IDM parameter.
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept.

Return type None

instantiation(*id*)

Get an Instantiation from a node id, usefull to fill the constraints of the network.

bnet accessors / shortcuts.

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

Returns the instantiation

Return type *pyAgrum.Instantiation* (page 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.

Doest 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 !

Return type None

`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 *pyAgrum.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 291) – If p<1

`setEpsilon`(*eps*)

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

Raises *pyAgrum.OutOfBounds* (page 291) – If eps<0

Return type None

`setMaxIter`(*max*)

Parameters **max** (*int*) – the maximum number of iteration

Raises *pyAgrum.OutOfBounds* (page 291) – If max <= 1

Return type None

`setMaxTime`(*timeout*)

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises *pyAgrum.OutOfBounds* (page 291) – 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 291) – 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 *pyAgrum.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 291) – 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 291) – If eps<0

Return type None

`setMaxIter(max)`

Parameters **max** (*int*) – the maximum number of iteration

Raises `pyAgrum.OutOfBounds` (page 291) – If max <= 1

Return type None

`setMaxTime(timeout)`

Parameters

- **tiemout** (*float*) – stopping criterion on timeout (in seconds)
- **timeout** (float) –

Raises `pyAgrum.OutOfBounds` (page 291) – 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 291) – 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`()

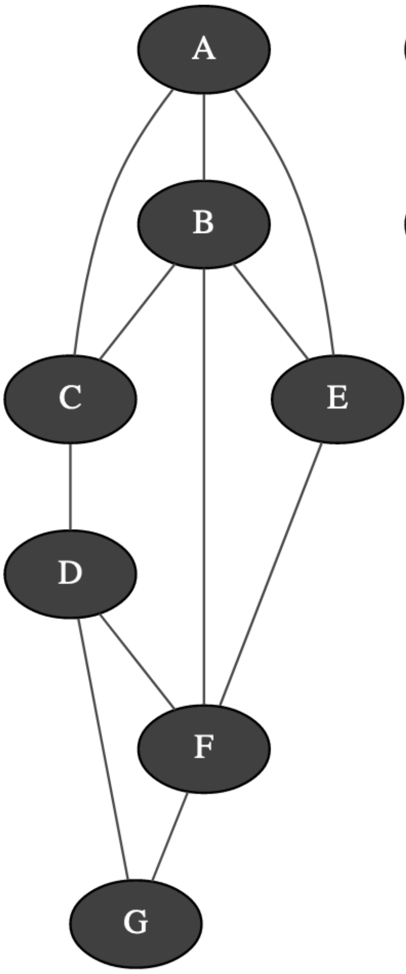
Returns True if the verbosity is enabled

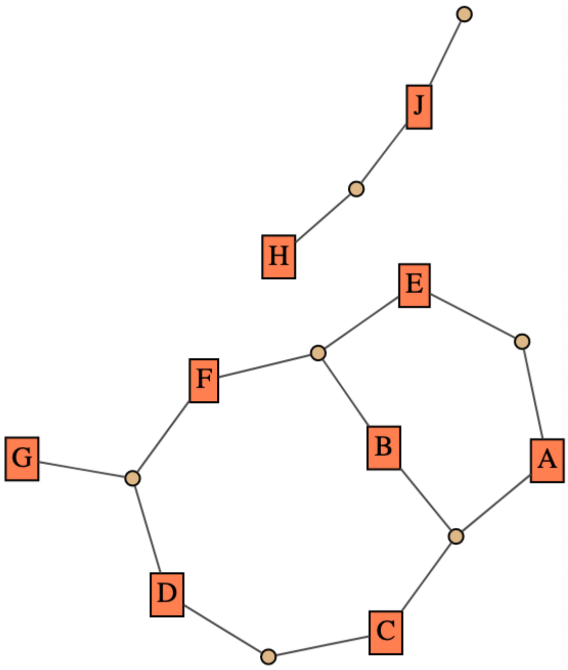
Return type bool

CHAPTER

SEVEN

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.

$$P(X_1, \cdots, X_n) \propto \prod_{i=1}^{n_c} \phi_i(C_i)$$

Where the ϕ_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/13_Models-MarkovNetwork.ipynb.html)

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 288) – 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)`

Return type *pyAgrum.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 288) – 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 *pyAgrum.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.

Returns

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** (`pyAgrum.BayesNet` (page 58)) –

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

`hasSameStructure(other)`

Parameters **other** (*pyAgrum.UGmodel*) –

Return type `bool`

`idFromName(name)`

Parameters **name** (`str`) –

Return type `int`

`ids(names)`

Parameters **names** (*Vector_string*) –

Return type pyAgrum.YetUnWrapped

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** (pyAgrum.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 *pyAgrum.DiscreteVariable* (page 25)

`variableFromName(name)`

Parameters **name** (`str`) –

Return type *pyAgrum.DiscreteVariable* (page 25)

`variableNodeMap()`

Return type pyAgrum.VariableNodeMap

7.2 Inference

Inference is the process that consists in computing new probabilistc 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.ShaferShenoyMNInference(`*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()`

Return type pyAgrum.IMarkovNet

`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 289) – If the node already has an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 289) – If the node does not already have an evidence
- *pyAgrum.InvalidArgument* (page 289) – If val is not a value for the node
- *pyAgrum.InvalidArgument* (page 289) – 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 292) – 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 292) – 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 292) – 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 instanciation 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 instanciation 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 evidene 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 292) – 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 292) – If node Id is not in the Bayesian network

`joinTree()`

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 13)

`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 bu 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 292) – 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 13)

`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 292) – 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

- **gum.InvalidArgument** – If one value is not a value for the node
- **gum.InvalidArgument** – If the size of a value is different from the domain side of the node
- **gum.FatalError** – If one value is a vector of 0s
- **gum.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 **gum.UndefinedElement** – 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

- `gum.InvalidArgument` – If one value is not a value for the node
- `gum.InvalidArgument` – If the size of a value is different from the domain side of the node
- `gum.FatalError` – If one value is a vector of 0s
- `gum.UndefinedElement` – If one node does not belong to the Bayesian network

CHAPTER

EIGHT

PROBABILISTIC RELATIONAL MODELS

For now, pyAgrum only allows to explore Probabilistic Relational Models written with o3prm syntax (see O3PRM website (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 interace extended by interface_name

Return type str

Raises `pyAgrum.IndexError` – If the interface is not in the PRM

`getSuperType`(*type_name*)

Parameters `type_name` (*str*) – a type name

Returns the type extended by type_name

Return type str

Raises `pyAgrum.IndexError` – If the type is not in the PRM

`getalltheSystems()`

Returns the list of all the systems and their components

Return type list

`interAttributes`(*interface_name*, *allAttributes=False*)

Parameters

- `interface_name` (*str*) – an interface
- `allAttributes` (*bool*) – True if supertypes of a custom type should be indicated

Returns the list of (<type>,<attribute_name>) for the given interface

Return type list

Raises `pyAgrum.IndexError` – If the type is not in the PRM

`interReferences`(*interface_name*)

Parameters `interface_name` (*str*) – an interface

Returns the list of (<reference_type>,<reference_name>,<True if the reference is an array>) for the given interface

Return type list

Raises `pyAgrum.IndexError` – If the type is not in the PRM

`interfaces()`

Returns the list of interfaces in the PRM

Return type list

`isAttribute`(*class_name*, *att_name*)

Parameters

- `class_name` (*str*) – a class name
- `att_name` (*str*) – the name of the attribute to be tested

Returns True if att_name is an attribute of class_name

Return type bool

Raises

- `pyAgrum.IndexError` – If the class is not in the PRM
- `pyAgrum.IndexError` – If att_name is not an element of class_name

`isClass`(*name*)

Parameters `name` (*str*) – an element name

Returns True if the parameter correspond to a class in the PRM

Return type bool

`isInterface(name)`

Parameters `name` (*str*) – an element name

Returns True if the parameter correspond to an interface in the PRM

Return type bool

`isType(name)`

Parameters `name` (*str*) – an element name

Returns True if the parameter correspond to a type in the PRM

Return type bool

`load(*args)`

Load a PRM into the explorer.

Parameters

- `filename` (*str*) – the name of the o3prm file
- `classpath` (*str*) – the classpath of the PRM

Raises `pyAgrum.FatalError` (page 288) – If file not found

Return type `None`

`types()`

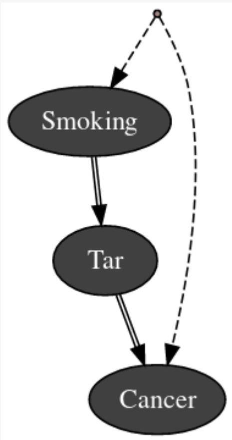
Returns the list of the custom types in the PRM

Return type list

CHAPTER

NINE

PYAGRUM.CAUSAL DOCUMENTATION



$$P(Cancer \mid \hookrightarrow Smoking) = \sum_{Tar} P\left(Tar \mid Smoking\right) \cdot \left(\sum_{Smoking'} P\left(Cancer \mid Smoking', Tar\right) \cdot P\left(Smoking'\right) \right)$$

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 abilidy 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.

Tutorial

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

Reference

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 wee 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 consits in

- a reference to the CausalModel
- Three sets of variables name that represent the 3 sets of variable in the query P(set1 | doing(set2),knowing(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 : $P(on|knowing, overhook (doing))$

Parameters

- **cm** (*CausalModel* (page 233)) – the causal model
- **root** (*ASTtree* (page 238)) – 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 233)
return: the causal model

Return type *CausalModel* (page 233)

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 238)**
return: ASTtree root of the CausalFormula tree

Return type `ASTtree` (page 238)

`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 233)) – 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** (`str`) – the variables of interest
- **cm** (`CausalModel` (page 233)) – 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 233)) – 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 238)`]`) – The ASTtree representing the calculus in construction

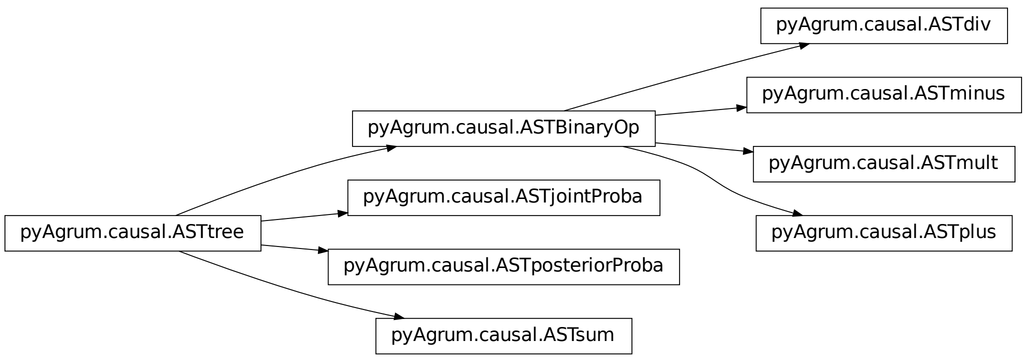
Return type *ASTtree* (page 238)

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 238).



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 238)

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 238)) – left operand
- **op2** (*ASTtree* (page 238)) – right operand
- **typ** (str) –

copy()
Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)
return: the left operand

Return type *ASTtree* (page 238)

property op2: *pyAgrum.causal._doAST.ASTtree* (page 238)
return: the right operand

Return type *ASTtree* (page 238)

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 238)) – first operand
- **op2** (*ASTtree* (page 238)) – second operand

copy()
Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)
return: the left operand

Return type *ASTtree* (page 238)

property op2: *pyAgrum.causal._doAST.ASTtree* (page 238)
return: the right operand

Return type *ASTtree* (page 238)

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 238)) – first operand
- **op2** (*ASTtree* (page 238)) – second operand

copy()
Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)**
return: the left operand

Return type *ASTtree* (page 238)

property op2: *pyAgrum.causal._doAST.ASTtree* **(page 238)**
return: the right operand

Return type *ASTtree* (page 238)

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 238)) – first operand
- **op2** (*ASTtree* (page 238)) – second operand

copy()
Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)
return: the left operand

Return type *ASTtree* (page 238)

property op2: *pyAgrum.causal._doAST.ASTtree* (page 238)
return: the right operand

Return type *ASTtree* (page 238)

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 238)) – first operand
- **op2** (*ASTtree* (page 238)) – second operand

`copy()`

Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)

return: the left operand

Return type *ASTtree* (page 238)

property op2: *pyAgrum.causal._doAST.ASTtree* (page 238)

return: the right operand

Return type *ASTtree* (page 238)

`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 238)) – the tree to be evaluated

copy()

Copy an CausalFormula tree

Return type *ASTtree* (page 238)

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 238)**

return: the ASTtree of the expression inside the sum

Return type *ASTtree* (page 238)

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 238)

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 238)

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 233) :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 intervention :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 233) :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 233) :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 intervention :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 233)) –
- **`size`** (`str`) –

CHAPTER TEN

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).

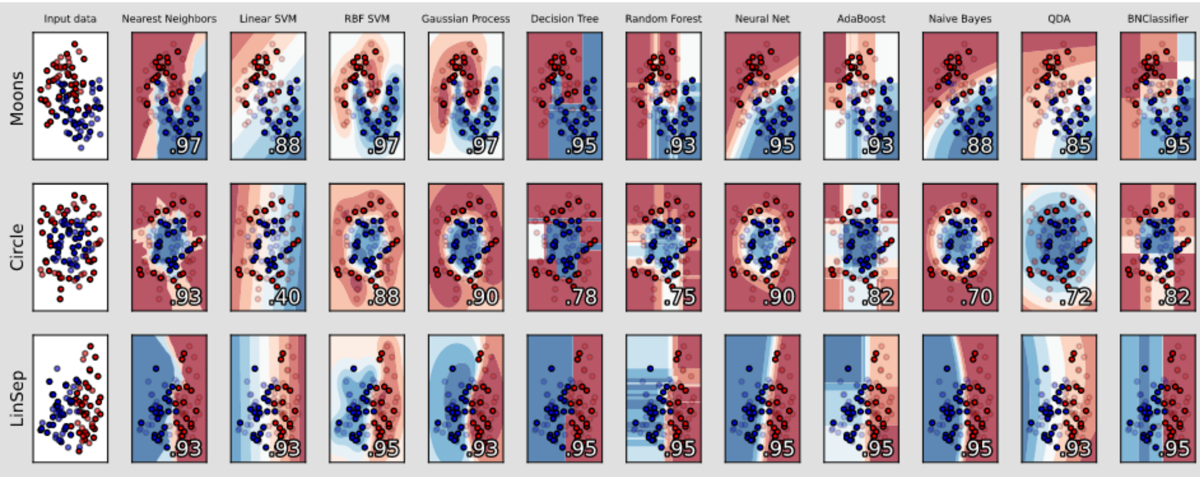


Fig. 1: An example from scikit-learn (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html) where a last column with a BNClassifier has been added flawlessly (see this notebook (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html)).

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), the integration in scikit-learn codes (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CompareClassifiersWithSklearn.ipynb.html) and, as an example, cross-validation with scikit-learn (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),
- Notebook on Discretizers in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Discretizer.ipynb.ht 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: PossibleEdges allows for both (tail,head) and (head,tail) to be added to the Bayesian network, while the others are not symmetric.

aPrioriWeight: double The weight used for a priori smoothing.

possibleSkeleton: pyagrum.undigraph An undirected graph that serves as a possible skeleton for the Bayesian network

DirichletCsv: str the file name of the csv file we want to use for the dirichlet prior. Will be ignored if 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 discetization method is used, this parameter is ignored.

discretizationThreshold: int or float When using default parameters a variable will be treated as continous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For instance, if you have entered 0.95, the variable will be treated as continous only if more than 95% of its values are unique.

usePR: bool indicates if the threshold to choose is Prevision-Recall curve's threhsold 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 soulb de 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 the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

discretizationThreshold: int or float When using default parameters a variable will be treated as continous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For example if you have entered 0.95, the variable will be treated as continous only if more than 95% of its values are unique.

`audit`(*X*, *y=None*)

parameters:

X: {array-like, sparse matrix} of shape (n_samples, n_features) training data

y: array-like of shape (n_samples,) Target values

returns: auditDict: dict()

Audits the passed values of X and y. Tells us which columns in X we think are already discrete and which need to be discretized, as well as the discretization algorithm that will be used to discretize them The parameters which are suggested will be used when creating the variables. To change this the user can manually set discretization parameters for each variable using the setDiscretizationParameters function.

`clear`(*clearDiscretizationParameters=False*)

parameters:

clearDiscretizationParamaters: bool if True, this method also clears the parameters the user has set for each variable and resets them to the default.

returns: void

Sets the number of continous variables and the total number of bins created by this discretizer to 0. If clearDiscretizationParameters is True, also clears the the parameters for discretization the user has set for each variable.

`createVariable`(*variableName*, *X*, *y=None*, *possibleValuesY=None*)

parameters:

variableName: the name of the created variable

X: ndarray shape(n,1) A column vector containing n samples of a feature. The column for which the variable will be created

y: ndarray shape(n,1) A column vector containing the corresponding for each element in X.

possibleValuesX: onedimensional ndarray An ndarray containing all the unique values of X

possibleValuesY: onedimensional ndarray An ndarray containing all the unique values of y

returnModifiedX: bool X could be modified by this function during

returns:

var: pyagrum.DiscreteVariable the created variable

Creates a variable for the column passed in as a parameter and places it in the Bayesian network

`discretizationCAIM`(*x*, *y*, *possibleValuesX*, *possibleValuesY*)

parametres:

x: ndarray with shape (n,1) where n is the number of samples Column-vector that contains all the data that needs to be discretized

y: ndarray with shape (n,1) where n is the number of samples Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

possibleValuesY: one dimensional ndarray Contains the possible values of y. There should be two possible values since this is a binary classifier

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Applies the CAIM algorithm to discretize the values of x

`discretizationElbowMethodRotation`(*discretizationStrategy*, *X*)

parameters:

discretizationStrategy: str The method of discretization that will be used. Possible values are: 'quantile' , 'kmeans' and 'uniform'

X: one dimensional ndarray Contains the data that should be discretized

returns: binEdges: the edges of the bins the algorithm has chosen.

Calculates the sum of squared errors as a function of the number of clusters using the discretization strategy that is passed as a parameter. Returns the bins that are optimal for minimizing the variation and the number of bins at the same time. Uses the elbow method to find this optimal point. To find the "elbow" we rotate the curve and look for its minimum.

`discretizationMDLP`(*x*, *y*, *possibleValuesX*, *possibleValuesY*)

parametres:

x: ndarray with shape (n,1) where n is the number of samples Column-vector that contains all the data that needs to be discretized

y: ndarray with shape (n,1) where n is the number of samples Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

possibleValuesY: one dimensional ndarray Contains the possible values of y. There should be two possible values since this is a binary classifier

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Uses the MDLP algorithm described in Fayyad, 1995 to discretize the values of x.

`discretizationNML`(*X*, *possibleValuesX*, *kMax=10*, *epsilon=None*)

parameters:

X: one dimensional ndarray array that that contains all the data that needs to be discretized

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside.

kMax: int the maximum number of bins before the algorithm stops itself.

epsilon: float or None the value of epsilon used in the algorithm. Should be as small as possible. If None is passed the value is automatically calculated.

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Uses the disceretization 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 paramaters of. Set to None to set the new default for this BNClassifier.

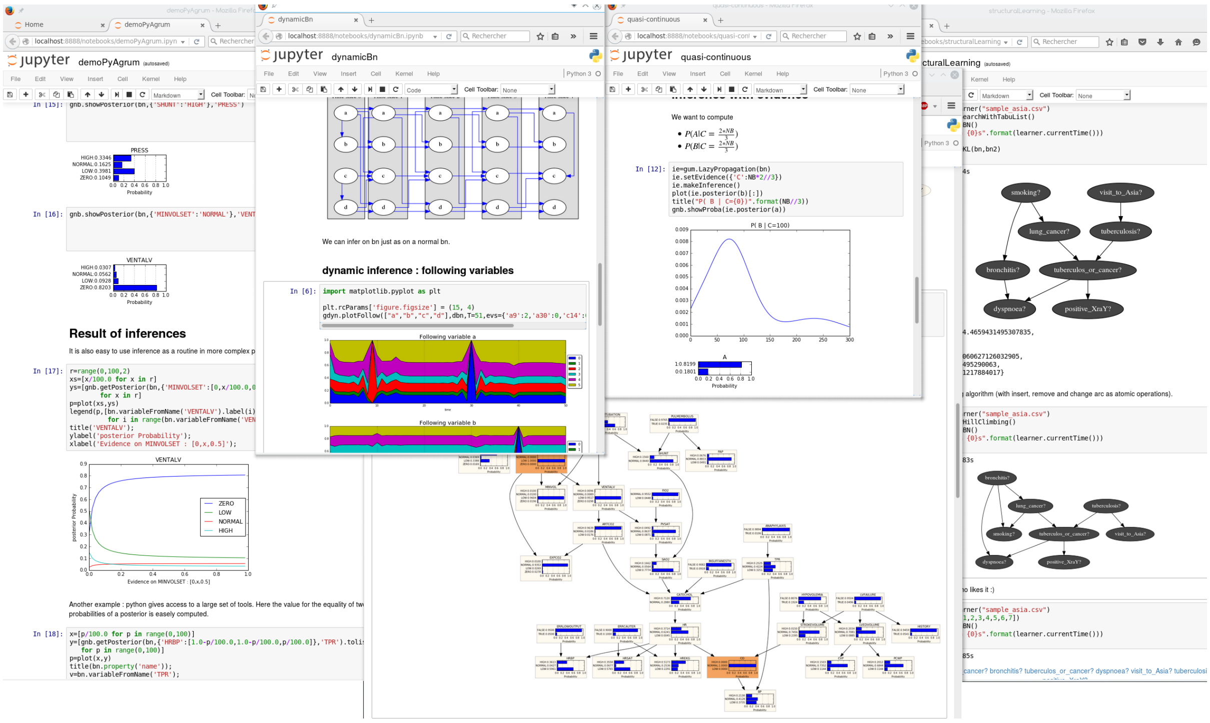
methode: str The method of discretization used for this variable. Type "NoDiscretization" if you do not want to discretize this variable. Possible values are: 'NoDiscretization', 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'

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 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).

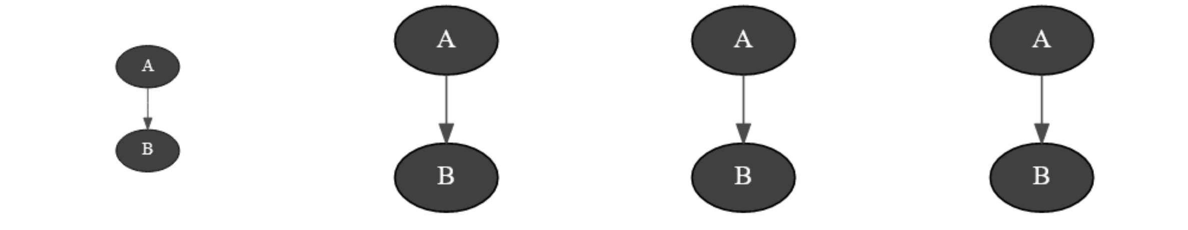


11.1 Visualization of graphical models

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 point "!" (such as *size="4!"*), then *size* is taken to be the desired minimum size. In this case, if both dimensions of the drawing are less than size, the drawing is scaled up uniformly until at least one dimension equals its dimension in size.

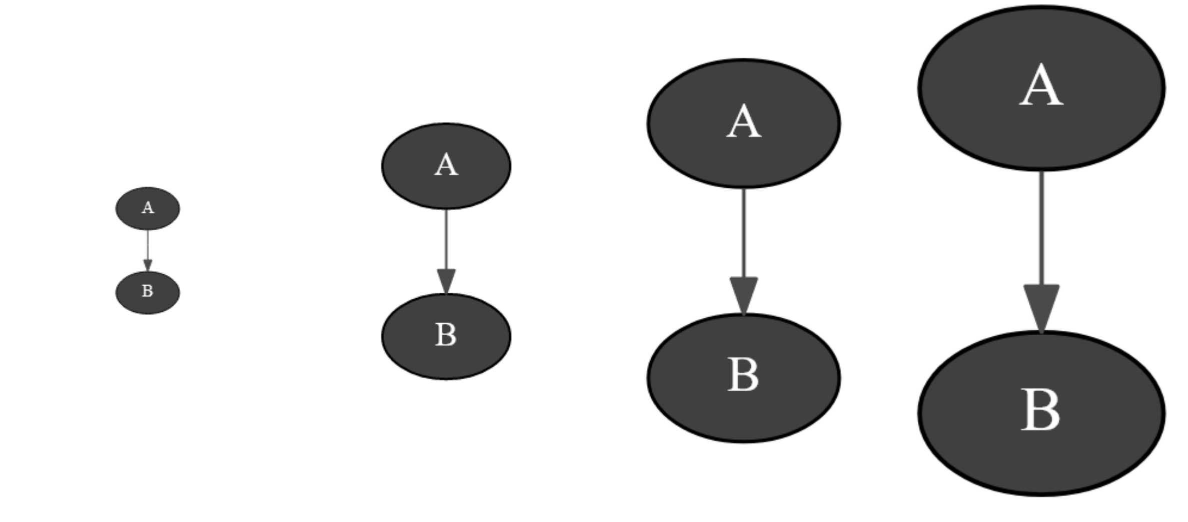
```
bn=gum.fastBN("A->B")
print("* withtout '!'")
gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}") for i in range(1,5)],captions=[f'size="{i}"' for i in range(1,5)])

print("* witht '!'")
gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}!") for i in range(1,5)],captions=[f'size="{i}!"' for i in range(1,5)])
```

```
* withtout '!'
```



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 (beeween 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 (beeween 0 and 1) to be shown as a color of factor. (used when view='factorgraph')
- **edgeWidth** – a edgeMap of values to be shown as width of edges (used when view='graph')
- **edgeColor** – a edgeMap of values (between 0 and 1) to be shown as color of edges (used when view='graph')
- **cmap** – color map to show the colors
- **cmapEdge** – color map to show the edge color if distinction is needed

Returns the graph

`pyAgrum.lib.notebook.showCN`(*cn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

show a credal network

Parameters

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

`pyAgrum.lib.notebook.getCN`(*cn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

get a HTML string for a credal network

Parameters

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

`pyAgrum.lib.notebook.showInference`(*model*, ***kwargs*)

show pydot graph for an inference in a notebook

Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

`pyAgrum.lib.notebook.getInference(model, **kwargs)`
get a HTML string for an inference in a notebook

Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

`pyAgrum.lib.notebook.`**`showJunctionTree`**`(`*bn*, *withNames=True*, *size=None*`)`
Show a junction tree

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 Exporting visualisations (as pdf,png)

11.4 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

11.5 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

11.6 Helpers

`pyAgrum.lib.notebook.configuration`()
Display the collection of dependance and versions

`pyAgrum.lib.notebook.sideBySide`(**args*, ***kwargs*)
display side by side args as HMTL fragment (using string, _repr_html_() or str())

Parameters

- **args** – HMTL fragments as string arg, arg._repr_html_() or str(arg)
- **captions** – list of strings (captions)

CHAPTER

TWELVE

PYAGRUM.LIB.IMAGE

pyAgrum.lib.image aims to graphically export models and inference using pydotplus (https://pypi.org/project/pydotplus/) (and then graphviz (https://graphviz.org/)).

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

```
import pyAgrum as gum
from pyAgrum.lib.image as gumimage

bn = gum.fastBN("a->b->d;a->c->d[3]->e;f->b")
gumimage.export(bn,"out/test_export.png",
                nodeColor={'a': 1,
                           'b': 0.3,
                           'c': 0.4,
                           'd': 0.1,
                           'e': 0.2,
                           'f': 0.5},
                arcColor={(0, 1): 0.2,
                          (1, 2): 0.5},
                arcWidth={(0, 3): 0.4,
                          (3, 2): 0.5,
                          (2,4) :0.6})
```

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** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

CHAPTER

THIRTEEN

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

Returns

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>*)

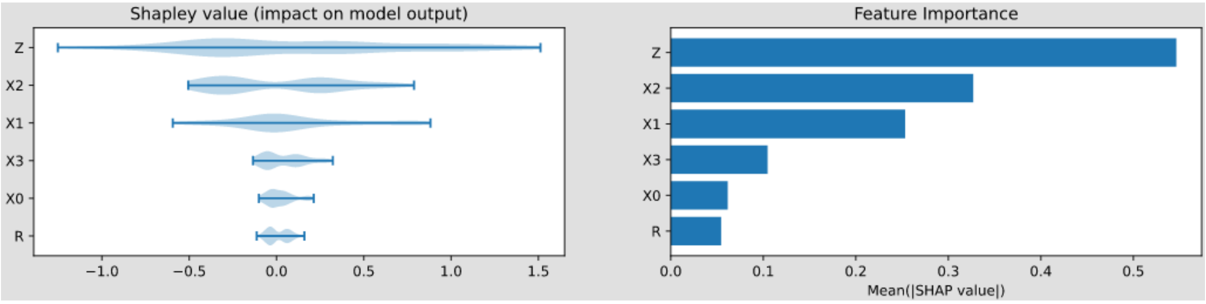
show a bn annotated with results from inference : entropy and mutual information

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[3], but the Interventional calculation method proposed in[2] is also present. In addition, a new causal method, based on[1], is implemented which is well suited for Bayesian networks.

bn [gum.BayesNet] The Bayesian network

target [str] the name of the target node

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.

Returns

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

[3] Lundberg, S. M., & Su-In, L. (2017). A Unified Approach to Interpreting Model. 31st Conference on Neural Information Processing Systems. Long Beach, CA, USA.

[2] Janzing, D., Minorics, L., & Blöbaum, P. (2019). Feature relevance quantification in explainable AI: A causality problem. arXiv: Machine Learning. Retrieved 6 24, 2021, from https://arxiv.org/abs/1910.13413

[1] Heskes, T., Sijben, E., Bucur, I., & Claassen, T. (2020). Causal Shapley Values: Exploiting Causal Knowledge. 34th Conference on Neural Information Processing Systems. Vancouver, Canada.

- **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.

Returns

Return type a dictionary Dict[str,float]

`marginal`(*train*, *sample_size=200*, *plot=False*, *plot_importance=False*, *percentage=False*)
Compute the marginal Shap Values for each variables.

Parameters

- **train** (*pandas.DataFrame*) – the database
- **sample_size** (*int*) – The computation of marginal ShapValue is very slow. The parameter allow to compute only on a fragment of the database.
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

Returns

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

Returns

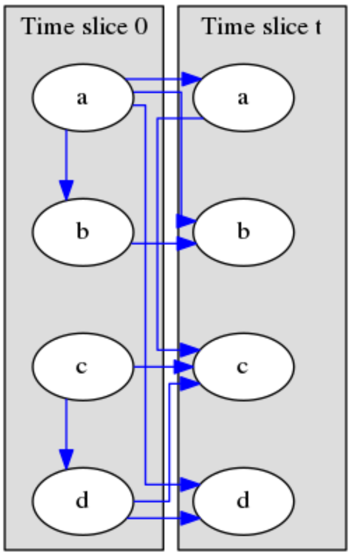
Return type a pydotplus.graph

PYAGRUM.LIB.DYNAMICBN

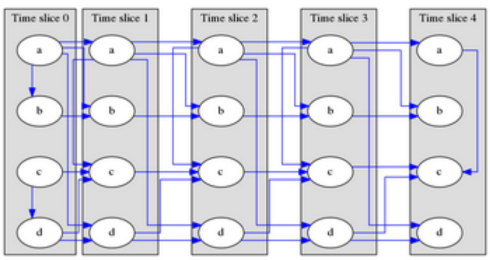
dynamic Bayesian Network are a special class of BNs where variables can be subscripted by a (discrete) time.

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

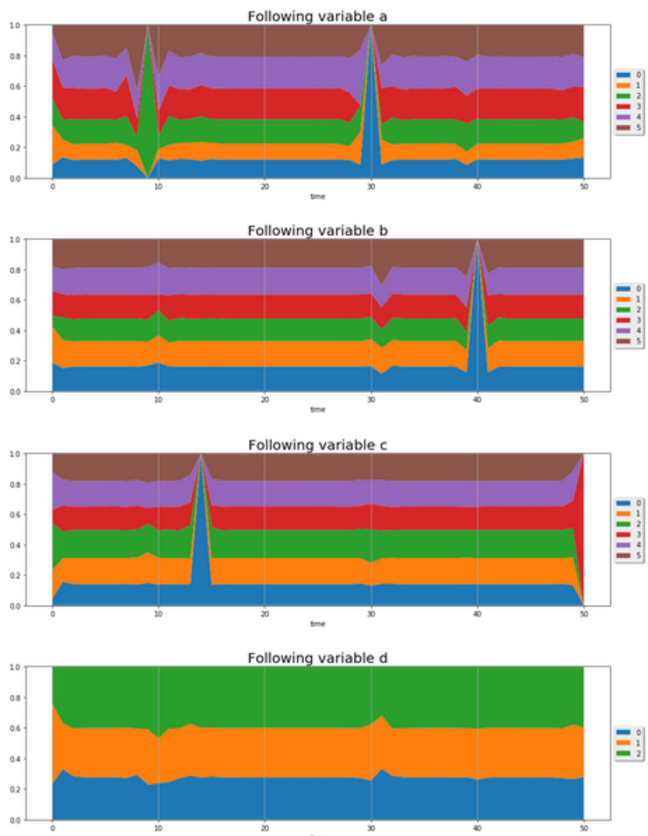
```
gdyn.showTimeSlices(dbn,format="png")
```




```
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 syntaxically 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*`)`
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

`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

CHAPTER

FIFTEEN

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 resuls ?
- **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 resuls
- **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 resuls
- **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 probabilitie

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, transforme_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 transforme_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 comaring different BNs.

`class pyAgrum.lib.bn_vs_bn.GraphicalBNComparator(name1, name2, delta=1e-06)`

Bases: `object`

BNGraphicalComparator allows to compare in multiple way 2 BNs…The smallest assumption is that the names of the variables are the same in the 2 BNs. But some comparisons will have also to check the type and domainSize of the variables. The bns have not exactly the same role : _bn1 is rather the referent model for the comparison whereas _bn2 is the compared one to the referent model.

Parameters

- **name1** (*str or* pyAgrum.BayesNet (page 58)) – a BN or a filename for reference
- **name2** (*str or* pyAgrum.BayesNet (page 58)) – another BN or antoher filename for comparison

dotDiff()

Return a pydotplus 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 pydotplus is not installed, this function just returns None

Returns the result dot graph or None if pydotplus can not be imported

Return type pydotplus.Dot

equivalentBNs()

Check if the 2 BNs are equivalent :

- same variables
- same graphical structure
- same parameters

Returns "OK" if bn are the same, a description of the error otherwise

Return type str

hamming()

Compute hamming and structural hamming distance

Hamming distance is the difference of edges comparing the 2 skeletons, and Structural Hamming difference is the difference comparing the cpdags, including the arcs' orientation.

Returns A dictionnary containing 'hamming','structural hamming'

Return type dict[double,double]

scores()

Compute Precision, Recall, F-score for self._bn2 compared to self._bn1

precision and recall are computed considering BN1 as the reference

Fscor is 2*(recall* precision)/(recall+precision) and is the weighted average of Precision and Recall.

dist2opt=square root of (1-precision)^2+(1-recall)^2 and represents the euclidian distance to the ideal point (precision=1, recall=1)

Returns A dictionnary 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

Fscor is 2*(recall* precision)/(recall+precision) and is the weighted average of Precision and Recall.

dist2opt=square root of (1-precision)^2+(1-recall)^2 and represents the euclidian distance to the ideal point (precision=1, recall=1)

Returns A dictionnary containing 'precision', 'recall', 'fscore', 'dist2opt' and so on.

Return type dict[str,double]

CHAPTER

SIXTEEN

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

Returns

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 specifiy 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/utiliy 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:
```

(continues on next page)

(continued from previous page)

```
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadBN('./bn.bif',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

pyAgrum.**saveBN**(*bn*, *filename*)

save a BN into a file using the format corresponding to one of the availableWriteBNExts() 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 availableMNExts() suffixes.

Listeners could be added in order to monitor its loading.

Examples

```
>>> import pyAgrum as gum
>>>
>>> # creating listeners
>>> def foo_listener(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
```

(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.loadMN('./bn.uai',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

`pyAgrum.saveMN(mn, filename)`
save a MN into a file using the format corresponding to one of the availableWriteMNExts() suffixes.

Parameters

- **mn(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 availableWriteIDExts() suffixes.

Parameters

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

CHAPTER

SEVENTEEN

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 liseners
>>> 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.

Returns

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

CHAPTER

EIGHTEEN

EXCEPTIONS FROM AGRUM

exception pyAgrum.**GumException**(**args*)

args

errorCallStack()

Returns the error call stack

Return type str

errorContent()

Returns the error content

Return type str

errorType()

Returns the error type

Return type str

what()

Return type `str`

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

All the exception classes inherit pyAgrum.GumException's functions errorType, errorCallStack and errorContent.

exception pyAgrum.**DefaultInLabel**(**args*)

args

property thisown

The membership flag

exception pyAgrum.**DuplicateElement**(**args*)

args

property thisown

The membership flag

what()

Return type str

exception pyAgrum.**DuplicateLabel**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**FatalError**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**FormatNotFound**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**GraphError**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**IOError**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidArc**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidArgument**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidArgumentsNumber**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidDirectedCycle**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidEdge**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**InvalidNode**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**NoChild**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**NoNeighbour**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**NoParent**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**NotFound**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**NullElement**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**OperationNotAllowed**(**args*)

args

property thisown
The membership flag

`what()`

Return type `str`

`exception pyAgrum.OutOfBounds(*args)`

`args`

`property thisown`
The membership flag

`what()`

Return type `str`

`exception pyAgrum.ArgumentError(*args)`

`args`

`property thisown`
The membership flag

`what()`

Return type `str`

`exception pyAgrum.SizeError(*args)`

`args`

`property thisown`
The membership flag

`what()`

Return type `str`

`exception pyAgrum.SyntaxError(*args)`

`args`

`col()`

Returns the indice of the colonne of the error

Return type int

`filename()`

Return type `str`

`line()`

Returns the indice of the line of the error

Return type int

`property thisown`
The membership flag

what()

Return type str

exception pyAgrum.**UndefinedElement**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**UndefinedIteratorKey**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**UndefinedIteratorValue**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**UnknownLabelInDatabase**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**DatabaseError**(**args*)

args

property thisown
The membership flag

what()

Return type str

exception pyAgrum.**CPTError**(**args*)

`args`

`property thisown`
The membership flag

`what()`

Return type `str`

CHAPTER

NINETEEN

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).

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 secion name or the property name does not exist

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X