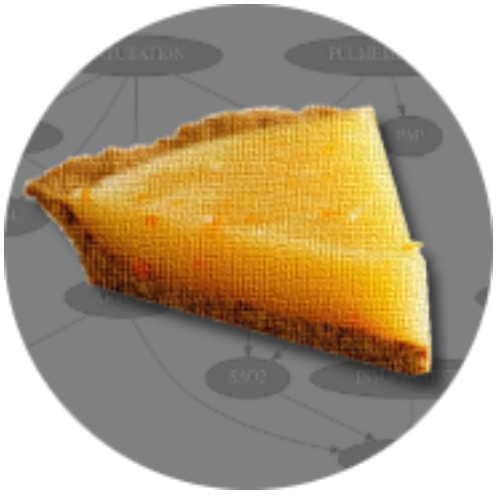


pyAgrum Documentation

Release 0.19.1

Pierre-Henri Wuillemin (Sphinx)

February 14, 2021

Fundamental components

1 Graphs manipulation **3**
- 1.1 Edges and Arcs . . . 3
- 1.2 Directed Graphs . . . 4
- 1.3 Undirected Graphs . . . 9
- 1.4 Mixed Graph . . . 16

2 Random Variables **21**
- 2.1 Common API for Random Discrete Variables . . . 21
- 2.2 Concrete classes for Random Discrete Variables . . . 23

3 Potential and Instantiation **33**
- 3.1 Instantiation . . . 34
- 3.2 Potential . . . 39

4 Bayesian network **47**
- 4.1 Model . . . 48
- 4.2 Tools for Bayesian networks . . . 60
- 4.3 Inference . . . 74
- 4.4 Exact Inference . . . 74
- 4.5 Approximated Inference . . . 94
- 4.6 Learning . . . 150

5 Markov Network **157**
- 5.1 Model . . . 158
- 5.2 Inference . . . 162

6 Influence Diagram **169**
- 6.1 Model . . . 170
- 6.2 Inference . . . 177

7 Probabilistic Relational Models **181**

8 Credal Network **187**
- 8.1 Model . . . 187
- 8.2 Inference . . . 192

9 pyAgrum.causal documentation **199**
- 9.1 Causal Model . . . 200
- 9.2 Causal Formula . . . 200
- 9.3 Causal Inference . . . 201
- 9.4 Abstract Syntax Tree for Do-Calculus . . . 202
- 9.5 Exceptions . . . 206

9.6 Notebook's tools for causality . . . 207

10 pyAgrum.skbn documentation **209**
10.1 Classifier using Bayesian networks . . . 210
10.2 Discretizer for Bayesian networks . . . 213

11 pyAgrum.lib.notebook **217**
11.1 Visualization of graphical models . . . 218
11.2 Visualization of Potentials . . . 218
11.3 Visualization of graphs . . . 218
11.4 Visualization of approximation algorithm . . . 218
11.5 Helpers . . . 218

12 Module bn2graph **219**
12.1 Visualization of Potentials . . . 219
12.2 Visualization of Bayesian networks . . . 220
12.3 Hi-level functions . . . 221

13 Module dynamic Bayesian network **223**

14 other pyAgrum.lib modules **225**
14.1 bn2roc . . . 225
14.2 bn2scores . . . 226
14.3 bn_vs_bn . . . 226
14.4 pretty_print . . . 228

15 Functions from pyAgrum **229**
15.1 Useful functions in pyAgrum . . . 229
15.2 Quick specification of (randomly parameterized) graphical models . . . 229
15.3 Input/Output for Bayesian networks . . . 231
15.4 Input/Output for Markov networks . . . 232
15.5 Input for influence diagram . . . 233

16 Other functions from aGrUM **235**
16.1 Listeners . . . 235
16.2 Random functions . . . 236
16.3 OMP functions . . . 237

17 Exceptions from aGrUM **239**

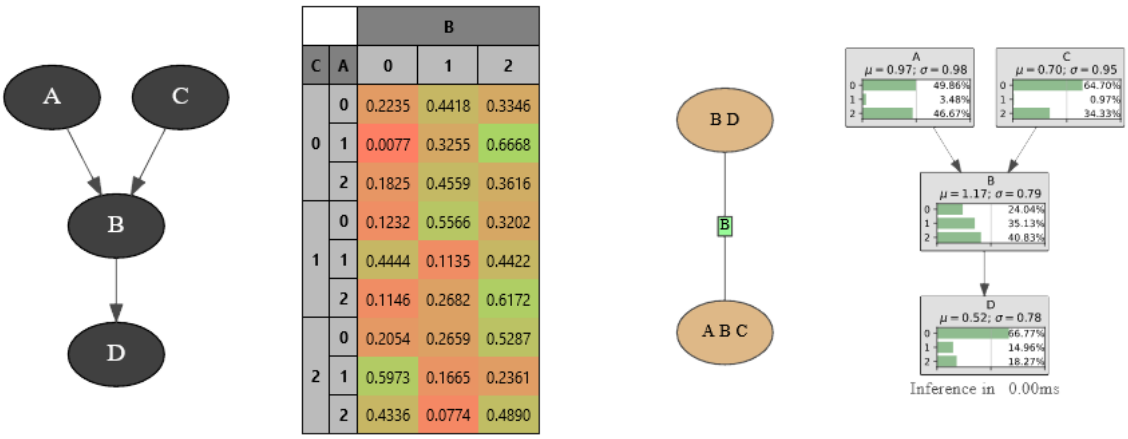
18 Configuration for pyAgrum **251**

19 Indices and tables **253**

Index **255**

pyAgrum (http://agrum.org) a scientific C++ and Python library dedicated to Bayesian networks 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 probabilsitic graphical models (Markov networks, influence diagrams and LIMIDs, dynamic BN, probabilistic relational models).

```
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'])
```



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

The module is generated using the SWIG (https://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/01-tutorial.ipynb.html),
- a gitlab repository (https://gitlab.com/agrumery/aGrUM),
- and a website (http://agrum.org).

CHAPTER 1

Graphs manipulation

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

For all types of graphs, nodes are int. If a graph of objects is needed (like *pyAgrum.BayesNet* (page 48)), 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 gum.Arc to copy

first(*Arc self*)

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

Return type int

head(*Arc self*)

Returns the id of the head node

Return type int

other(*Arc self*, *int id*)

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

Returns the nodeId of the other node

Return type int

second (*Arc self*)

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

Return type int

tail (*Arc self*)

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 (*Edge self*)

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

Return type int

other (*Edge self*, *int 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 (*Edge self*)

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 (*DiGraph self*, *int tail*, *int head*)
addArc(DiGraph self, int n1, int 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

Raises gum.InvalidNode – If head or tail does not belong to the graph nodes.

addNode (*DiGraph self*)

Returns the new NodeId

Return type int

addNodeWithId (*DiGraph self*, *int id*)
Add a node by choosing a new NodeId.

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

Raises gum.DuplicateElement – If the given id is already used

addNodes (*DiGraph self*, *int n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

arcs (*DiGraph self*)

Returns the list of the arcs

Return type List

children (*DiGraph self*, *int id*)

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

Returns the set of all the children

Return type Set

clear (*DiGraph self*)
Remove all the nodes and arcs from the 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])

empty (*DiGraph self*)
Check if the graph is empty.

Returns True if the graph is empty

Return type bool

emptyArcs (*DiGraph self*)
Check if the graph doesn't contains arcs.

Returns True if the graph doesn't contains arcs

Return type bool

eraseArc (*DiGraph self*, *int n1*, *int 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

eraseChildren (*DiGraph self*, *int n*)
Erase the arcs heading through the node's children.

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

eraseNode (*DiGraph self*, *int id*)
Erase the node and all the related arcs.

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

eraseParents (*DiGraph self*, *int n*)
Erase the arcs coming to the node.

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

existsArc (*DiGraph self*, *int n1*, *int 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 (*DiGraph self*, *int 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 (*DiGraph self*, *int _from*, *int 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

Returns True if the directed path exists

Return type bool

nodes (*DiGraph self*)

Returns the set of ids

Return type set

parents (*DiGraph self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

size(*DiGraph self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*DiGraph self*)

Returns the number of arcs in the graph

Return type int

toDot(*DiGraph self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*DiGraph self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises `gum.InvalidDirectedCycle` – If this graph contains cycles

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(*DAG self*, *int tail*, *int head*)

addArc(DAG self, int n1, int 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

Raises

- `gum.InvalidDirectedCircle` – If any (directed) cycle is created by this arc
- `gum.InvalidNode` – If head or tail does not belong to the graph nodes

addNode(*DiGraph self*)

Returns the new NodeId

Return type int

addNodeWithId(*DiGraph self*, *int id*)

Add a node by choosing a new NodeId.

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

Raises `gum.DuplicateElement` – If the given id is already used

addNodes (*DiGraph self*, *int n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

ancestors (*DAG self*, *int id*)

arcs (*DiGraph self*)

Returns the list of the arcs

Return type List

children (*DiGraph self*, *int id*)

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

Returns the set of all the children

Return type Set

clear (*DiGraph self*)
Remove all the nodes and arcs from the 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])

descendants (*DAG self*, *int id*)

empty (*DiGraph self*)
Check if the graph is empty.

Returns True if the graph is empty

Return type bool

emptyArcs (*DAG self*)

eraseArc (*DAG self*, *int n1*, *int n2*)

eraseChildren (*DAG self*, *int n*)

eraseNode (*DiGraph self*, *int id*)
Erase the node and all the related arcs.

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

eraseParents (*DAG self*, *int n*)

existsArc (*DAG self*, *int n1*, *int n2*)

existsNode (*DiGraph self*, *int 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 (*DiGraph self*, *int _from*, *int 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

Returns True if the directed path exists

Return type bool

isIndependent (*DAG self*, *int X*, *int Y*, *Set Z*)
isIndependent(DAG self, Set X, Set Y, Set Z) -> bool

moralGraph (*DAG self*)

moralizedAncestralGraph (*DAG self*, *Set nodes*)

nodes (*DiGraph self*)

Returns the set of ids

Return type set

parents (*DiGraph self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

size (*DiGraph self*)

Returns the number of nodes in the graph

Return type int

sizeArcs (*DAG self*)

toDot (*DiGraph self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder (*DiGraph self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises `gum.InvalidDirectedCycle` – If this graph contains cycles

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(*UndiGraph self*, *int first*, *int second*)
addEdge(UndiGraph self, int n1, int 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 gum.InvalidNode – If n1 or n2 does not belong to the graph nodes.

addNode(*UndiGraph self*)

Returns the new NodeId

Return type int

addNodeWithId(*UndiGraph self*, *int id*)
Add a node by choosing a new NodeId.

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

Raises gum.DuplicateElement – If the given id is already used

addNodes(*UndiGraph self*, *int n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

clear(*UndiGraph self*)
Remove all the nodes and edges from the 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])

edges(*UndiGraph self*)

Returns the list of the edges

Return type List

empty(*UndiGraph self*)
Check if the graph is empty.

Returns True if the graph is empty

Return type bool

emptyEdges(*UndiGraph self*)
Check if the graph doesn't contains edges.

Returns True if the graph doesn't contains edges

Return type bool

eraseEdge(*UndiGraph self*, *int n1*, *int 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

eraseNeighbours(*UndiGraph self*, *int n*)
Erase all the edges adjacent to a given node.

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

eraseNode(*UndiGraph self*, *int id*)
Erase the node and all the adjacent edges.

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

existsEdge(*UndiGraph self*, *int n1*, *int 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(*UndiGraph self*, *int 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(*UndiGraph self*)
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

neighbours(*UndiGraph self*, *int id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*UndiGraph self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*UndiGraph self*)

partialUndiGraph(*UndiGraph self*, *Set nodes*)

Parameters **nodesSet** (*Set*) – The set of nodes composing the partial graph

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

size(*UndiGraph self*)

Returns the number of nodes in the graph

Return type int

sizeEdges(*UndiGraph self*)

Returns the number of edges in the graph

Return type int

toDot(*UndiGraph self*)

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(*CliqueGraph self*, *int first*, *int 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

Raises gum.InvalidNode – If n1 or n2 does not belong to the graph nodes.

addNode(*CliqueGraph self*, *Set clique*)

addNode(CliqueGraph self) -> int addNode(CliqueGraph self, int id, Set clique) addNode(CliqueGraph self, int id)

Returns the new NodeId

Return type int

addNodeWithId(*UndiGraph self*, *int id*)

Add a node by choosing a new NodeId.

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

Raises gum.DuplicateElement – If the given id is already used

addNodes(*UndiGraph self*, *int n*)

Add n nodes.

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

Returns the new ids

Return type Set of int

addToClique(*CliqueGraph self*, *int clique_id*, *int 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

- gum.NotFound – If clique_id does not exist
- gum.DuplicateElement – If clique_id set already contains the ndoe

clear(*CliqueGraph self*)
Remove all the nodes and edges from the graph.

clearEdges(*CliqueGraph self*)
Remove all edges and their separators

clique(*CliqueGraph self*, *int clique*)

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

Returns The set of nodes included in the clique

Return type Set

Raises gum.NotFound – 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(*CliqueGraph self*, *int idNode*)

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

Returns the id of a clique containing the node

Return type int

Raises gum.NotFound – If no clique contains idNode

containerPath(*CliqueGraph self*, *int node1*, *int 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 gum.NotFound – If such path cannot be found

edges(*UndiGraph self*)

Returns the list of the edges

Return type List

empty(*UndiGraph self*)
Check if the graph is empty.

Returns True if the graph is empty

Return type bool

emptyEdges(*UndiGraph self*)
Check if the graph doesn't contains edges.

Returns True if the graph doesn't contains edges

Return type bool

eraseEdge(*CliqueGraph self*, *Edge 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

eraseFromClique(*CliqueGraph self*, *int clique_id*, *int 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 gum.NotFound – If clique_id does not exist

eraseNeighbours(*UndiGraph self*, *int n*)
Erase all the edges adjacent to a given node.

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

eraseNode(*CliqueGraph self*, *int node*)
Erase the node and all the adjacent edges.

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

existsEdge(*UndiGraph self*, *int n1*, *int 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(*UndiGraph self*, *int 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(*CliqueGraph self*)

Returns True if the running intersection property holds

Return type bool

hasUndirectedCycle(*UndiGraph self*)
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

isJoinTree(*CliqueGraph self*)

Returns True if the graph is a join tree

Return type bool

neighbours(*UndiGraph self*, *int id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*UndiGraph self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*UndiGraph self*)

partialUndiGraph(*UndiGraph self*, *Set nodes*)

Parameters **nodesSet** (*Set*) – The set of nodes composing the partial graph

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

separator(*CliqueGraph self*, *int cliq1*, *int 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

Returns the separator included in a given edge

Return type Set

Raises gum.NotFound – If the edge does not belong to the clique graph

setClique(*CliqueGraph self*, *int idClique*, *Set 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 gum.NotFound – If idClique is not a clique of the graph

size(*UndiGraph self*)

Returns the number of nodes in the graph

Return type int

sizeEdges(*UndiGraph self*)

Returns the number of edges in the graph

Return type int

toDot(*CliqueGraph self*)

Returns a friendly display of the graph in DOT format

Return type str

toDotWithNames(*bn*)

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) –
- **Bayesian network** (*a*) –

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(*MixedGraph self*, *int n1*, *int 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

Raises gum.InvalidNode – If head or tail does not belong to the graph nodes.

addEdge(*MixedGraph self*, *int n1*, *int 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 gum.InvalidNode – If n1 or n2 does not belong to the graph nodes.

addNode(*MixedGraph self*)

Returns the new NodeId

Return type int

addNodeWithId(*MixedGraph self*, *int id*)

Add a node by choosing a new NodeId.

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

Raises gum.DuplicateElement – If the given id is already used

addNodes(*MixedGraph self*, *int n*)

Add n nodes.

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

Returns the new ids

Return type Set of int

arcs(*DiGraph self*)

Returns the list of the arcs

Return type List

children(*DiGraph self*, *int id*)

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

Returns the set of all the children

Return type Set

clear(*MixedGraph self*)

Remove all the nodes and edges from the 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])

edges(*UndiGraph self*)

Returns the list of the edges

Return type List

empty(*MixedGraph self*)
Check if the graph is empty.

Returns True if the graph is empty

Return type bool

emptyArcs(*MixedGraph self*)
Check if the graph doesn't contains arcs.

Returns True if the graph doesn't contains arcs

Return type bool

emptyEdges(*MixedGraph self*)
Check if the graph doesn't contains edges.

Returns True if the graph doesn't contains edges

Return type bool

eraseArc(*MixedGraph self*, *int n1*, *int 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

eraseChildren(*MixedGraph self*, *int n*)
Erase the arcs heading through the node's children.

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

eraseEdge(*MixedGraph self*, *int n1*, *int 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

eraseNeighbours(*MixedGraph self*, *int n*)
Erase all the edges adjacent to a given node.

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

eraseNode(*MixedGraph self*, *int id*)
Erase the node and all the related arcs and edges.

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

eraseParents (*MixedGraph self*, *int n*)
Erase the arcs coming to the node.

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

existsArc (*MixedGraph self*, *int n1*, *int 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 (*MixedGraph self*, *int n1*, *int 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 (*MixedGraph self*, *int 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 (*DiGraph self*, *int _from*, *int 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

Returns True if the directed path exists

Return type bool

hasUndirectedCycle (*UndiGraph self*)
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

mixedOrientedPath (*MixedGraph self*, *int node1*, *int 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)

Return type List

Raises `gum.NotFound` – If no path can be found between the two nodes

mixedUnorientedPath(*MixedGraph self*, *int node1*, *int 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)

Return type List

Raises gum.NotFound – If no path can be found between the two nodes

neighbours(*UndiGraph self*, *int id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*UndiGraph self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*UndiGraph self*)

parents(*DiGraph self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

partialUndiGraph(*UndiGraph self*, *Set nodes*)

Parameters **nodesSet** (*Set*) – The set of nodes composing the partial graph

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

size(*MixedGraph self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*MixedGraph self*)

Returns the number of arcs in the graph

Return type int

sizeEdges(*MixedGraph self*)

Returns the number of edges in the graph

Return type int

toDot(*MixedGraph self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*DiGraph self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises `gum.InvalidDirectedCycle` – If this graph contains cycles

CHAPTER 2

Random Variables

aGrUM/pyAgrum is currently dedicated for discrete probability distributions.

There are 3 types of discrete random variables in aGrUM/pyAgrum: LabelizedVariable, DiscretizedVariable and RangeVariable. The 3 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 base class for discrete random variables.

DiscreteVariable(aName, aDesc='') -> DiscreteVariable

Parameters:

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

DiscreteVariable(aDRV) -> DiscreteVariable

Parameters:

- **aDRV** (*pyAgrum.DiscreteVariable*) – the pyAgrum.DiscreteVariable that will be copied

description(*Variable self*)

Returns the description of the variable

Return type str

domain(*DiscreteVariable self*)

Returns the domain of the variable

Return type str

domainSize(*DiscreteVariable self*)

Returns the number of modalities in the variable domain

Return type int

empty(*DiscreteVariable self*)

Returns True if the domain size < 2

Return type bool

index(*DiscreteVariable self*, *str label*)

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

Returns the indice of the label

Return type int

label(*DiscreteVariable self*, *int i*)

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

Returns the indice-th label

Return type str

Raises gum.OutOfBound – If the variable does not contain the label

labels(*DiscreteVariable self*)

Returns a tuple containing the labels

Return type tuple

name(*Variable self*)

Returns the name of the variable

Return type str

numerical(*DiscreteVariable self*, *int indice*)

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

Returns the numerical representation of the indice-th value

Return type float

setDescription(*Variable self*, *str theValue*)
set the description of the variable.

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

setName(*Variable self*, *str theValue*)
sets the name of the variable.

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

toDiscretizedVar(*DiscreteVariable self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

Raises gum.RuntimeError – If the variable is not a DiscretizedVariable

toLabelizedVar(*DiscreteVariable self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

Raises gum.RuntimeError – If the variable is not a LabelizedVariable

toRangeVar(*DiscreteVariable self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

Raises gum.RuntimeError – If the variable is not a RangeVariable

toStringWithDescription(*DiscreteVariable self*)

Returns a description of the variable

Return type str

varType(*DiscreteVariable self*)
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<0,1,2>
>>>
>>> va.addLabel('foo')
>>> print(va)
>>> ## a<0,1,2,foo>
>>>
>>> va.chgLabel(1,'bar')
>>> print(va)
>>> a<0,bar,2,foo>
>>>
>>> vb=gum.LabelizedVariable('b','b',0).addLabel('A').addLabel('B').addLabel('C
↪')
>>> print(vb)
>>> ## b<A,B,C>
>>>
>>> vb.labels()
>>> ## ('A', 'B', 'C')
>>>
>>> vb.isLabel('E')
>>> ## False
```

(continues on next page)

(continued from previous page)

```
>>>
>>> vb.label(2)
>>> ## 'B'
```

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 23)

Raises gum.DuplicateElement – If the variable already contains the label

changeLabel (*LabelizedVariable self*, *int pos*, *str 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

- gum.DuplicatedElement – If the variable already contains the new label
- gum.OutOfBounds – If the index is greater than the size of the variable

description (*Variable self*)

Returns the description of the variable

Return type str

domain (*LabelizedVariable self*)

Returns the domain of the variable as a string

Return type str

domainSize (*LabelizedVariable self*)

Returns the number of modalities in the variable domain

Return type int

empty (*DiscreteVariable self*)

Returns True if the domain size < 2

Return type bool

eraseLabels (*LabelizedVariable self*)
Erase all the labels from the variable.

index (*LabelizedVariable self*, *str label*)

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

Returns the indice of the label

Return type int

isLabel (*LabelizedVariable self*, *str 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 (*LabelizedVariable self*, *int i*)

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

Returns the indice-th label

Return type str

Raises gum.OutOfBound – If the variable does not contain the label

labels (*DiscreteVariable self*)

Returns a tuple containing the labels

Return type tuple

name (*Variable self*)

Returns the name of the variable

Return type str

numerical (*LabelizedVariable self*, *int indice*)

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

Returns the numerical representation of the indice-th value

Return type float

posLabel (*LabelizedVariable self*, *str label*)

setDescription (*Variable self*, *str theValue*)
set the description of the variable.

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

setName (*Variable self*, *str theValue*)
sets the name of the variable.

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

toDiscretizedVar (*DiscreteVariable self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

Raises gum.RuntimeError – If the variable is not a DiscretizedVariable

toLabelizedVar (*DiscreteVariable self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

Raises gum.RuntimeError – If the variable is not a LabelizedVariable

toRangeVar (*DiscreteVariable self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

Raises gum.RuntimeError – If the variable is not a RangeVariable

toStringWithDescription (*DiscreteVariable self*)

Returns a description of the variable

Return type str

varType (*LabelizedVariable self*)
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')
>>> vX.addTick(1).addTick(2).addTick(3).addTick(3.1415) #doctest: +ELLIPSIS
>>> ## <pyAgrum.DiscretizedVariable;...>
>>> print(vX)
>>> ## X<[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** (*double*) – the Tick to be added

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

Raises `gum.DefaultInLabel` – If the tick is already defined

description(*Variable self*)

Returns the description of the variable

Return type str

domain(*DiscretizedVariable self*)

Returns the domain of the variable as a string

Return type str

domainSize(*DiscretizedVariable self*)

Returns the number of modalities in the variable domain

Return type int

empty(*DiscreteVariable self*)

Returns True if the domain size < 2

Return type bool

eraseTicks(*DiscretizedVariable self*)
erase all the Ticks

index(*DiscretizedVariable self*, *str label*)

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

Returns the indice of the label

Return type int

isTick(*DiscretizedVariable self*, *double aTick*)

Parameters **aTick** (*double*) – the Tick to be tested

Returns True if the Tick already exists

Return type bool

label(*DiscretizedVariable self*, *int i*)

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

Returns the indice-th label

Return type str

Raises gum.OutOfBound – If the variable does not contain the label

labels(*DiscreteVariable self*)

Returns a tuple containing the labels

Return type tuple

name(*Variable self*)

Returns the name of the variable

Return type str

numerical(*DiscretizedVariable self*, *int indice*)

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

Returns the numerical representation of the indice-th value

Return type float

setDescription(*Variable self*, *str theValue*)
set the description of the variable.

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

setName(*Variable self*, *str theValue*)
sets the name of the variable.

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

tick(*DiscretizedVariable self*, *int i*)
Indicate the index of the Tick

Parameters **i** (*int*) – the index of the Tick

Returns **aTick** – the index-th Tick

Return type double

Raises `gum.NotFound` – If the index is greater than the number of Ticks

ticks (*DiscretizedVariable self*)

Returns a tuple containing all the Ticks

Return type tuple

toDiscretizedVar (*DiscreteVariable self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toLabelizedVar (*DiscreteVariable self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar (*DiscreteVariable self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

Raises `gum.RuntimeError` – If the variable is not a RangeVariable

toStringWithDescription (*DiscreteVariable self*)

Returns a description of the variable

Return type str

varType (*DiscretizedVariable self*)
returns the type of variable

Returns the type of the variable, 0: DiscretizedVariable, 1: LabelizedVariable, 2: RangeVariable

Return type int

2.2.3 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.gum.RangeVariable('I','I in [4,10]',4,10)
>>> print(vI)
>>> ## I[4-10]
>>>
>>> vX.maxVal()
>>> ## 10
>>>
>>> vX.belongs(1)
>>> ## False
>>>
>>> # where is the value 5 ?
>>> vX.index('5')
>>> ## 1
>>>
>>> vi.labels()
>>> ## ('4', '5', '6', '7', '8', '9', '10')
```

belongs (*RangeVariable self*, *long val*)

Parameters **val** (*long*) – the value to be tested

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

Return type bool

description (*Variable self*)

Returns the description of the variable

Return type str

domain (*RangeVariable self*)

Returns the domain of the variable

Return type str

domainSize (*RangeVariable self*)

Returns the number of modalities in the variable domain

Return type int

empty (*DiscreteVariable self*)

Returns True if the domain size < 2

Return type bool

index (*RangeVariable self*, *str arg2*)

Parameters **arg2** (*str*) – a label

Returns the indice of the label

Return type int

label (*RangeVariable self*, *int indice*)

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

Returns the indice-th label

Return type str

Raises gum.OutOfBound – If the variable does not contain the label

labels (*DiscreteVariable self*)

Returns a tuple containing the labels

Return type tuple

maxVal (*RangeVariable self*)

Returns the upper bound of the variable.

Return type long

minVal (*RangeVariable self*)

Returns the lower bound of the variable

Return type long

name (*Variable self*)

Returns the name of the variable

Return type str

numerical (*RangeVariable self*, *int indice*)

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

Returns the numerical representation of the indice-th value

Return type float

setDescription (*Variable self*, *str theValue*)
set the description of the variable.

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

setMaxVal (*RangeVariable self*, *long maxVal*)
Set a new value of the upper bound

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

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

setMinVal (*RangeVariable self*, *long minVal*)
Set a new value of the lower bound

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

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

setName (*Variable self*, *str theValue*)
sets the name of the variable.

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

toDiscretizedVar (*DiscreteVariable self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toLabelizedVar(*DiscreteVariable self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar(*DiscreteVariable self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

Raises `gum.RuntimeError` – If the variable is not a RangeVariable

toStringWithDescription(*DiscreteVariable self*)

Returns a description of the variable

Return type str

varType(*RangeVariable self*)
returns the type of variable

Returns the type of the variable, 0: DiscretizedVariable, 1: LabelizedVariable, 2: RangeVariable

Return type int

CHAPTER 3

Potential and Instantiation

pyAgrum.Potential (page 39) is a multi-dimensional array with a *pyAgrum.DiscreteVariable* (page 21) 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 39) with the sequence of *pyAgrum.DiscreteVariable* (page 21) 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 39) 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("{} -> {}".format(I,p.get(I)))
<A:1|B:1> -> 0.16666666666666666
>>> I["B"]="2";print("{} -> {}".format(I,p.get(I)))
<A:1|B:2> -> 0.26666666666666666
```

- *pyAgrum.Potential* (page 39) include tensor operators (see for instance this notebook (http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/05-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
>>> 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(*Instantiation self*, *DiscreteVariable v*)

Adds a new variable in the Instantiation.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The new variable added to the Instantiation

Raises *DuplicateElement* (page 239) – If the variable is already in this Instantiation

chgVal(*Instantiation self*, *DiscreteVariable v*, *int newval*)

chgVal(Instantiation self, DiscreteVariable v, int newval) -> Instantiation chgVal(Instantiation self, int varPos, int newval) -> Instantiation chgVal(Instantiation self, str var, int newval) -> Instantiation chgVal(Instantiation self, str var, str newval) -> Instantiation

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

Parameters

- **v** (pyAgrum.DiscreteVariable (page 21) *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 34)

Raises

- *NotFound* (page 246) – If variable v does not belong to the instantiation.
- OutOfBound – If newval is not a possible value for the variable.

clear (*Instantiation self*)
Erase all variables from an Instantiation.

contains (*Instantiation self*, *DiscreteVariable v*)
contains(Instantiation self, str name) -> bool contains(Instantiation self, DiscreteVariable v) -> bool

Indicates whether a given variable belongs to the Instantiation.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable for which the test is made.

Returns True if the variable is in the Instantiation.

Return type bool

dec (*Instantiation self*)
Operator –.

decIn (*Instantiation self*, *Instantiation i*)
Operator – for the variables in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The set of variables to decrement in this Instantiation

decNotVar (*Instantiation self*, *DiscreteVariable v*)
Operator – for vars which are not v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable not to decrement in this Instantiation.

decOut (*Instantiation self*, *Instantiation i*)
Operator – for the variables not in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The set of variables to not decrement in this Instantiation.

decVar (*Instantiation self*, *DiscreteVariable v*)
Operator – for variable v only.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable to decrement in this Instantiation.

Raises *NotFound* (page 246) – If variable v does not belong to the Instantiation.

domainSize (*Instantiation self*)

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

Return type int

empty (*Instantiation self*)

Returns True if the instantiation is empty.

Return type bool

end (*Instantiation self*)

Returns True if the Instantiation reached the end.

Return type bool

erase (*Instantiation self*, *DiscreteVariable v*)
erase(Instantiation self, str name)

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable to be removed from this Instantiation.

Raises *NotFound* (page 246) – If v does not belong to this Instantiation.

fromdict (*Instantiation self*, *PyObject * dict*)
Change the values in an instantiation from a dict (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.

hamming (*Instantiation self*)

Returns the hamming distance of this instantiation.

Return type int

inOverflow (*Instantiation self*)

Returns True if the current value of the tuple is correct

Return type bool

inc (*Instantiation self*)
Operator ++.

incIn (*Instantiation self*, *Instantiation i*)
Operator ++ for the variables in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The set of variables to increment in this Instantiation.

incNotVar (*Instantiation self*, *DiscreteVariable v*)
Operator ++ for vars which are not v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable not to increment in this Instantiation.

incOut (*Instantiation self*, *Instantiation i*)
Operator ++ for the variables not in i.

Parameters **i** (Instantiation (page 34)) – The set of variable to not increment in this Instantiation.

incVar (*Instantiation self*, *DiscreteVariable v*)
Operator ++ for variable v only.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable to increment in this Instantiation.

Raises *NotFound* (page 246) – If variable v does not belong to the Instantiation.

isMutable (*Instantiation self*)

nbrDim (*Instantiation self*)

Returns The number of variables in the Instantiation.

Return type int

pos (*Instantiation self*, *DiscreteVariable v*)

Returns the position of the variable v.

Return type int

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – the variable for which its position is return.

Raises *NotFound* (page 246) – If v does not belong to the instantiation.

rend(*Instantiation self*)

Returns True if the Instantiation reached the rend.

Return type bool

reorder(*Instantiation self*, *pyAgrum.Sequence< pyAgrum.DiscreteVariable * > v*)
reorder(Instantiation self, Instantiation i)

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

Parameters

- **i** (pyAgrum.Instantiation (page 34)) – The sequence of variables with which to reorder this Instantiation.
- **v** (*list*) – The new order of variables for this Instantiation.

setFirst(*Instantiation self*)
Assign the first values to the tuple of the Instantiation.

setFirstIn(*Instantiation self*, *Instantiation i*)
Assign the first values in the Instantiation for the variables in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The variables to which their first value is assigned in this Instantiation.

setFirstNotVar(*Instantiation self*, *DiscreteVariable v*)
Assign the first values to variables different of v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable that will not be set to its first value in this Instantiation.

setFirstOut(*Instantiation self*, *Instantiation i*)
Assign the first values in the Instantiation for the variables not in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The variable that will not be set to their first value in this Instantiation.

setFirstVar(*Instantiation self*, *DiscreteVariable v*)
Assign the first value in the Instantiation for var v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable that will be set to its first value in this Instantiation.

setLast(*Instantiation self*)
Assign the last values in the Instantiation.

setLastIn(*Instantiation self*, *Instantiation i*)
Assign the last values in the Instantiation for the variables in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The variables to which their last value is assigned in this Instantiation.

setLastNotVar(*Instantiation self*, *DiscreteVariable v*)
Assign the last values to variables different of v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable that will not be set to its last value in this Instantiation.

setLastOut(*Instantiation self*, *Instantiation i*)
Assign the last values in the Instantiation for the variables not in i.

Parameters **i** (pyAgrum.Instantiation (page 34)) – The variables that will not be set to their last value in this Instantiation.

setLastVar (*Instantiation self*, *DiscreteVariable v*)
Assign the last value in the Instantiation for var v.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable that will be set to its last value in this Instantiation.

setMutable (*Instantiation self*)

setVals (*Instantiation self*, *Instantiation i*)
Assign the values from i in the Instantiation.

Parameters **i** (pyAgrum.Instantiation (page 34)) – An Instantiation in which the new values are searched

Returns a reference to the instantiation

Return type *pyAgrum.Instantiation* (page 34)

todict (*Instantiation self*, *bool withLabels=False*)
Create a dict (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

unsetEnd (*Instantiation self*)
Alias for unsetOverflow().

unsetOverflow (*Instantiation self*)
Removes the flag overflow.

val (*Instantiation self*, *int i*)
val(Instantiation self, DiscreteVariable var) -> int val(Instantiation self, str name) -> int

Parameters

- **i** (*int*) – The index of the variable.
- **var** (pyAgrum.DiscreteVariable (page 21)) – The variable the value of which we wish to know

Returns the current value of the variable.

Return type int

Raises *NotFound* (page 246) – If the element cannot be found.

variable (*Instantiation self*, *int i*)
variable(Instantiation self, str name) -> DiscreteVariable

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

Returns the variable at position i in the tuple.

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises *NotFound* (page 246) – If the element cannot be found.

variablesSequence (*Instantiation self*)

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(*Potential self*, *Potential p*)

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

Parameters **p** (pyAgrum.Potential (page 39)) – the potential from which we want to calculate the divergence.

Returns The value of the divergence

Return type float

Raises

- gum.InvalidArgument – If p is not compatible with the potential (dimension, variables)
- gum.FatalError – If a zero is found in p or the potential and not in the other.

abs(*Potential self*)

Apply abs on every element of the container

Returns a reference to the modified potential.

Return type *pyAgrum.Potential* (page 39)

add(*Potential self*, *DiscreteVariable v*)

Add a discrete variable to the potential.

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – the var to be added

Raises

- *DuplicateElement* (page 239) – If the variable is already in this Potential.
- *InvalidArgument* (page 243) – If the variable is empty.

argmax(*Potential self*)

argmin(*Potential self*)

contains(*Potential self*, *DiscreteVariable v*)

Parameters **v** (pyAgrum.Potential (page 39)) – a DiscreteVariable.

Returns True if the var is in the potential

Return type bool

domainSize(*Potential self*)

draw(*Potential self*)

draw a value using the potential as a probability table.

Returns the index of the drawn value

Return type int

empty(*Potential self*)

Returns Returns true if no variable is in the potential.

Return type bool

entropy (*Potential self*)

Returns the entropy of the potential

Return type double

extract (*Potential self*, *Instantiation inst*)
extract(Potential self, PyObject * dict) -> Potential

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

Parameters

- **inst** (*pyAgrum.instantiation*) – a partial instantiation
- **dict** (*dict*) – a dictionnary containing discrete variables (?)

Returns the new Potential

Return type *pyAgrum.Potential* (page 39)

fillWith (*Potential self*, *Potential src*)
fillWith(Potential self, Potential src, Vector_string mapSrc) -> Potential fillWith(Potential self, Vector v) -> Potential fillWith(Potential self, double v) -> Potential

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 39)

Raises `gum.SizeError` – 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 39)

Raises `gum.InvalidArgument` – If the first variable is Labelized or if the len of the noise is not odd.

findAll (*Potential self*, *double v*)

get (*Potential self*, *Instantiation i*)

Parameters **i** (`pyAgrum.Instantiation` (page 34)) – an Instantiation

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

Return type double

inverse (*Potential self*)

isNonZeroMap (*Potential self*)

Returns a boolean-like potential using the predicate isNonZero

Return type *pyAgrum.Potential* (page 39)

log2 (*Potential self*)
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.

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 (*Potential self*, *PyObject * varnames*)
Projection using max as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

margMaxOut (*Potential self*, *PyObject * varnames*)
Projection using max as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

Raises `gum.InvalidArgument` – If varnames contains only one variable that does not exist in the Potential

margMinIn (*Potential self*, *PyObject * varnames*)
Projection using min as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

margMinOut (*Potential self*, *PyObject * varnames*)
Projection using min as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

margProdIn (*Potential self*, *PyObject * varnames*)
Projection using multiplication as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

margProdOut (*Potential self*, *PyObject * varnames*)
Projection using multiplication as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

Raises `gum.InvalidArgument` – If varnames contains only one variable that does not exist in the Potential

margSumIn (*Potential self*, *PyObject * varnames*)
Projection using sum as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

margSumOut (*Potential self*, *PyObject * varnames*)
Projection using sum as operation.

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

Raises `gum.InvalidArgument` – If varnames contains only one variable that does not exist in the Potential

max (*Potential self*)

Returns the maximum of all elements in the Potential

Return type double

maxNonOne (*Potential self*)

Returns the maximum of non one elements in the Potential

Return type double

Raises `gum.NotFound` – If all value == 1.0

min (*Potential self*)

Returns the min of all elements in the Potential

Return type double

minNonZero (*Potential self*)

Returns the min of non zero elements in the Potential

Return type double

Raises `gum.NotFound` – If all value == 0.0

nbrDim (*Potential self*)
nbrDim(Potential self) -> int

Returns the number of vars in the multidimensional container.

Return type int

newFactory (*Potential self*)
Erase the Potential content and create a new empty one.

Returns a reference to the new Potential

Return type *pyAgrum.Potential* (page 39)

new_abs (*Potential self*)

new_log2 (*Potential self*)

new_sq (*Potential self*)

noising (*Potential self*, *double alpha*)

normalize (*Potential self*)
Normalize the Potential (do nothing if sum is 0)

Returns a reference to the normalized Potential

Return type *pyAgrum.Potential* (page 39)

normalizeAsCPT (*Potential self*, *int varId=0*)
Normalize the Potential as a CPT

Returns a reference to the normalized Potential

Return type *pyAgrum.Potential* (page 39)

Raises `gum.FatalError` – If some distribution sums to 0

pos (*Potential self*, *DiscreteVariable v*)

Parameters **v** (`pyAgrum.DiscreteVariable` (page 21)) – The variable for which the index is returned.

Returns

Return type Returns the index of a variable.

Raises `gum.NotFound` – If v is not in this multidimensional matrix.

product (*Potential self*)

Returns the product of all elements in the Potential

Return type double

putFirst(*Potential self*, *str varname*)

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable for which the index should be 0.

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

Raises gum.InvalidArgument – If the var is not in the potential

random(*Potential self*)

randomCPT(*Potential self*)

randomDistribution(*Potential self*)

remove(*Potential self*, *DiscreteVariable var*)

Parameters **v** (pyAgrum.DiscreteVariable (page 21)) – The variable to be removed

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

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

reorganize(*Potential self*, *vector< pyAgrum.DiscreteVariable *, allocator< pyAgrum.DiscreteVariable * > > vars*)

reorganize(Potential self, Vector_string vars) -> Potential

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 39)

scale(*Potential self*, *double v*)

Create a new potential multiplied by v.

Parameters **v** (*double*) – a multiplier

Returns

Return type a reference to the modified potential

set(*Potential self*, *Instantiation i*, *double value*)

Change the value pointed by i

Parameters

- **i** (pyAgrum.Instantiation (page 34)) – The Instantiation to be changed
- **value** (*double*) – The new value of the Instantiation

sq(*Potential self*)

Square all the values in the Potential

sum(*Potential self*)

Returns the sum of all elements in the Potential

Return type double

toarray()

Returns the potential as an array

Return type array

tolist()

Returns the potential as a list

Return type list

translate(*Potential self*, *double v*)
Create a new potential added with v.

Parameters **v** (*double*) – The value to be added

Returns

Return type a reference to the modified potential

var_dims

Returns a list containing the dimensions of each variables in the potential

Return type list

var_names

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

Return type list

> **Warning:** Listed in reverse from the variable enumeration order

variable(*Potential self*, *int i*)
variable(Potential self, str name) -> DiscreteVariable

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

Returns

Return type the varible at the ith index

Raises gum.NotFound – If i does not reference a variable in this multidimensional matrix.

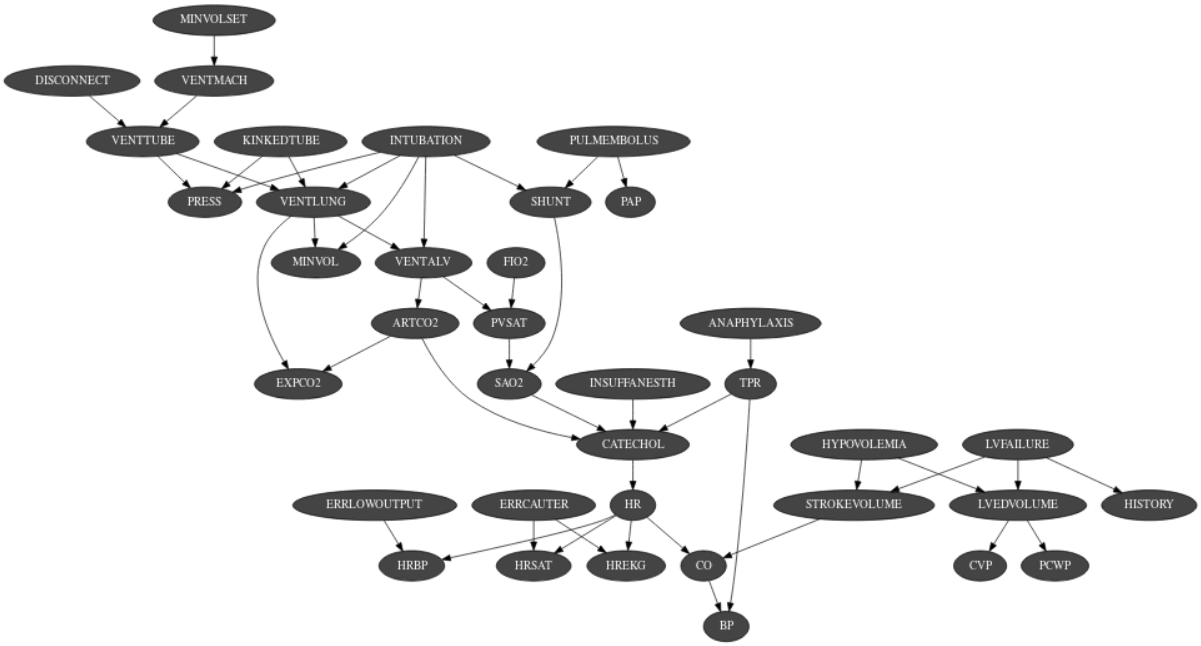
variablesSequence()

Returns a list containing the sequence of variables

Return type list

CHAPTER 4

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/01-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(*BayesNet self*, *DiscreteVariable var*)

add(BayesNet self, str name, unsigned int nbrmod) -> int add(BayesNet self, DiscreteVariable var, pyAgrum.MultiDimImplementation aContent) -> int add(BayesNet self, DiscreteVariable var, int id) -> int add(BayesNet self, DiscreteVariable var, pyAgrum.MultiDimImplementation aContent, int id) -> int

Add a variable to the pyAgrum.BayesNet.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – 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

- gum.DuplicateLabel – If variable.name() is already used in this pyAgrum.BayesNet.
- gum.NotAllowed – If nbrmod is less than 2
- gum.DuplicateElement – If id is already used.

addAMPLITUDE(*BayesNet self*, *DiscreteVariable var*)

Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addAND(*BayesNet self*, *DiscreteVariable 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 21)) – The variable added by copy.

Returns the id of the added variable.

Return type int

Raises `gum.SizeError` – If variable.domainSize()>2

addArc(*BayesNet self*, *int tail*, *int head*)
addArc(BayesNet self, str tail, str head)

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

- `gum.InvalidEdge` – If arc.tail and/or arc.head are not in the BN.
- `gum.DuplicateElement` – If the arc already exists.

addCOUNT(*BayesNet self*, *DiscreteVariable var*, *int value=1*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addEXISTS(*BayesNet self*, *DiscreteVariable var*, *int value=1*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addFORALL(*BayesNet self*, *DiscreteVariable var*, *int value=1*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added variable.

Return type int

addLogit(*BayesNet self*, *DiscreteVariable var*, *double external_weight*, *int id*)
addLogit(BayesNet self, DiscreteVariable var, double external_weight) -> int

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 21)) – The variable added by copy
- **externalWeight** (*double*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

Returns the id of the added variable.

Return type int

Raises `gum.DuplicateElement` – If id is already used

addMAX(*BayesNet self*, *DiscreteVariable var*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addMEDIAN(*BayesNet self*, *DiscreteVariable var*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addMIN(*BayesNet self*, *DiscreteVariable var*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

Returns the id of the added value

Return type int

addNoisyAND(*BayesNet self*, *DiscreteVariable var*, *double external_weight*, *int id*)
addNoisyAND(BayesNet self, DiscreteVariable var, double external_weight) -> int

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 21)) – The variable added by copy
- **externalWeight** (*double*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

Returns the id of the added variable.

Return type int

Raises `gum.DuplicateElement` – If id is already used

addNoisyOR(*BayesNet self*, *DiscreteVariable var*, *double external_weight*)
addNoisyOR(BayesNet self, DiscreteVariable var, double external_weight, int id) -> int

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 21)) – The variable added by copy
- **externalWeight** (*double*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

Returns the id of the added variable.

Return type int

Raises `gum.DuplicateElement` – If id is already used

addNoisyORCompound(*BayesNet self*, *DiscreteVariable var*, *double external_weight*)
addNoisyORCompound(BayesNet self, DiscreteVariable var, double external_weight, int id) -> int

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 21)) – The variable added by copy
- **externalWeight** (*double*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

Returns the id of the added variable.

Return type int

Raises `gum.DuplicateElement` – If id is already used

addNoisyORNet(*BayesNet self*, *DiscreteVariable var*, *double external_weight*)
addNoisyORNet(BayesNet self, DiscreteVariable var, double external_weight, int id) -> int

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 21)) – The variable added by copy
- **externalWeight** (*double*) – the added external weight
- **id** (*int*) – The proposed id for the variable.

Returns the id of the added variable.

Return type int

addOR(*BayesNet self*, *DiscreteVariable 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 21)) – The variable added by copy

Returns the id of the added variable.

Return type int

Raises `gum.SizeError` – If variable.domainSize()>2

addSUM(*BayesNet self*, *DiscreteVariable var*)
Others aggregators

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable to be added

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(*BayesNet self*, *int tail*, *int head*, *double causalWeight*)
addWeightedArc(BayesNet self, str tail, str head, double causalWeight)

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** (*double*) – the added causal weight

Raises

- gum.InvalidArc – If arc.tail and/or arc.head are not in the BN.
- gum.InvalidArc – If variable in arc.head is not a NoisyOR variable.

ancestors(*BayesNet self*, *PyObject * norid*)

arcs(*BayesNet self*)

Returns The lisf of arcs in the IBayesNet

Return type list

beginTopologyTransformation(*BayesNet self*)
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.

changePotential(*BayesNet self*, *int id*, *Potential newPot*)
changePotential(BayesNet self, str name, Potential newPot)

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

Parameters

- **newPot** (pyAgrum.Potential (page 39)) – the new potential
- **NodeId** (*int*) – the id of the node

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

Raises gum.NotAllowed – If newPot has not the same signature as __probaMap[NodeId]

changeVariableLabel(*BayesNet self*, *int id*, *str old_label*, *str new_label*)
changeVariableLabel(BayesNet self, str name, str old_label, str new_label)

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 gum.NotFound – if id/name is not a variable or if old_label does not exist.

changeVariableName(*BayesNet self*, *int id*, *str new_name*)
changeVariableName(BayesNet self, str name, str new_name)

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

- gum.DuplicateLabel – If new_name is already used in this BayesNet.
- gum.NotFound – If no variable matches id.

children(*BayesNet self*, *PyObject * norid*)

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

Returns the set of all the children

Return type Set

clear(*BayesNet self*)
Clear the whole BayesNet

completeInstantiation(*GraphicalModel self*)

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(*BayesNet self*, *int varId*)
cpt(BayesNet self, str name) -> Potential

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 39)

Raises `gum.NotFound` – If no variable's id matches varId.

dag (*BayesNet self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

descendants (*BayesNet self*, *PyObject * norid*)

dim (*IBayesNet self*)
Returns the dimension (the number of free parameters) in this BayesNet.

Returns the dimension of the BayesNet

Return type int

empty (*GraphicalModel self*)

endTopologyTransformation (*BayesNet self*)
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 48)

erase (*BayesNet self*, *int varId*)
erase(BayesNet self, str name) erase(BayesNet self, DiscreteVariable var)

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 21)) – A reference on the variable to remove.

eraseArc (*BayesNet self*, *Arc arc*)
eraseArc(BayesNet self, int tail, int head) eraseArc(BayesNet self, str tail, str head)

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)

exists (*DAGmodel self*, *int node*)

existsArc (*DAGmodel self*, *int tail*, *int head*)
existsArc(DAGmodel self, str nametail, str namehead) -> bool

family (*DAGmodel self*, *int id*)
family(DAGmodel self, str name) -> Set

static fastPrototype (*str dotlike*, *int 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 gum.RangeVariable using the default domain size (second argument)
  - with 'a[10]', the variable is a gum.RangeVariable using 10 as domain size (from 0 to 9)
  - with 'a[3,7]', the variable is a gum.RangeVariable using a domainSize from 3 to 7
  - with 'a[1,3.14,5,6.2]', the variable is a gum.DiscretizedVariable using the given ticks (at least 3 values)
  - with 'a{top|middle|bottom}', the variable is a gum.LabelizedVariable using the given labels.

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=gum.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 48)

generateCPT (*BayesNet self*, *int node*)
generateCPT(BayesNet self, str name)

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

Parameters

- **node** (*int*) – The variable's id.
- **name** (*str*) – The variable's name.

generateCPTs (*BayesNet self*)
Randomly generates CPTs for a given structure.

hasSameStructure (*DAGmodel self*, *DAGmodel other*)

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

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

Return type bool

idFromName(*BayesNet self*, *str 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 gum.NotFound – If name does not match a variable in the graph

ids(*GraphicalModel self*, *Vector_string names*)

isIndependent(*BayesNet self*, *PyObject * X*, *PyObject * Y*, *PyObject * Z*)

jointProbability(*IBayesNet self*, *Instantiation i*)

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

Returns a parameter of the joint probability for the BayesNet

Return type double

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

loadBIF(*BayesNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load a BIF file.

Parameters

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

loadBIFXML(*BayesNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load a BIFXML file.

Parameters

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

loadDSL(*BayesNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load a DSL file.

Parameters

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

Raises

- gum.IOError – If file not found

- gum.FatalError – If file is not valid

loadNET(*BayesNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load a NET file.

Parameters

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

loadO3PRM(*BayesNet self*, *str name*, *str system=""*, *str classpath=""*, *PyObject * l=(PyObject *) 0*)
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

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

loadUAI(*BayesNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load an UAI file.

Parameters

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

log10DomainSize(*BayesNet self*)

log2JointProbability(*IBayesNet self*, *Instantiation i*)

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

Returns a parameter of the log joint probability for the BayesNet

Return type double

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

maxNonOneParam(*IBayesNet self*)

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

Return type double

maxParam(*IBayesNet self*)

Returns the biggest value in the CPTs of the BayesNet

Return type double

maxVarDomainSize(*IBayesNet self*)

Returns the biggest domain size among the variables of the BayesNet

Return type int

minNonZeroParam(*IBayesNet self*)

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

Return type double

minParam(*IBayesNet self*)

Returns the smallest value in the CPTs of the IBayesNet

Return type double

minimalCondSet(*BayesNet self*, *int target*, *PyObject * list*)

minimalCondSet(BayesNet self, PyObject * targets, PyObject * list) -> PyObject *

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(*DAGmodel self*, *bool 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 9)

moralizedAncestralGraph(*BayesNet self*, *PyObject * nodes*)

names(*BayesNet self*)

Returns The names of the graph variables

Return type list

nodeId(*BayesNet self*, *DiscreteVariable var*)

Parameters **var** (pyAgrum.DiscreteVariable (page 21)) – a variable

Returns the id of the variable

Return type int

Raises gum.IndexError – If the graph does not contain the variable

nodes(*BayesNet self*)

Returns the set of ids

Return type set

nodeset(*GraphicalModel self*, *Vector_string names*)

parents(*BayesNet self*, *PyObject * norid*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

property(*GraphicalModel self*, *str name*)

propertyWithDefault(*GraphicalModel self*, *str name*, *str byDefault*)

reverseArc(*BayesNet self*, *int tail*, *int head*)
reverseArc(BayesNet self, str tail, str head) reverseArc(BayesNet self, Arc arc)

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 gum.InvalidArc – If the arc does not exsit or if its reversal would induce a directed cycle.

saveBIF(*BayesNet self*, *str name*)
Save the BayesNet in a BIF file.

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

saveBIFXML(*BayesNet self*, *str name*)
Save the BayesNet in a BIFXML file.

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

saveDSL(*BayesNet self*, *str name*)
Save the BayesNet in a DSL file.

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

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

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

saveO3PRM(*BayesNet self*, *str 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

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

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

setProperty(*GraphicalModel self*, *str name*, *str value*)

size (*BayesNet self*)

Returns the number of nodes in the graph

Return type int

sizeArcs (*DAGmodel self*)

Returns the number of arcs in the graph

Return type int

toDot (*IBayesNet self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder (*DAGmodel self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises `gum.InvalidDirectedCycle` – If this graph contains cycles

variable (*BayesNet self*, *int id*)
variable(BayesNet self, str name) -> DiscreteVariable

Parameters

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises `gum.IndexError` – If the graph does not contain the variable

variableFromName (*BayesNet self*, *str name*)

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises `gum.IndexError` – If the graph does not contain the variable

variableNodeMap (*BayesNet self*)

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.

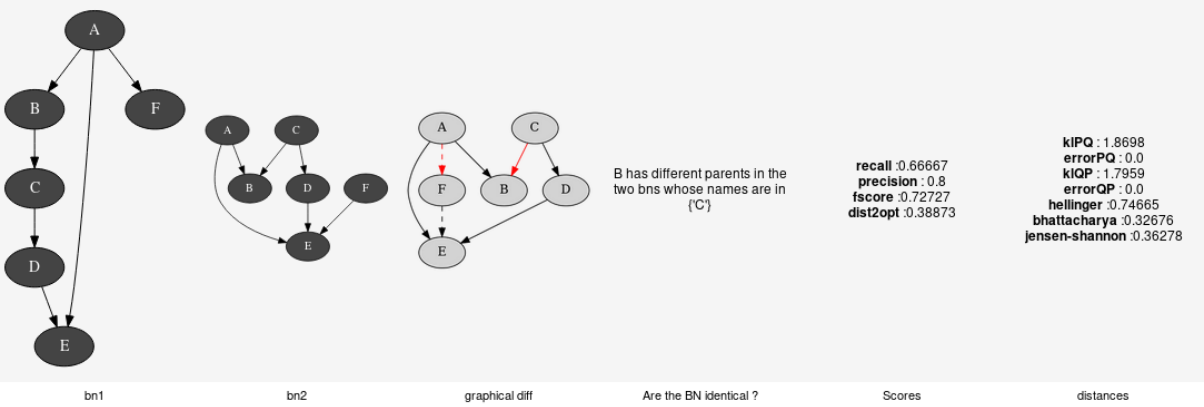
4.2.1 Generation of database

class pyAgrum.**BNDatabaseGenerator** (*bn: pyAgrum.BayesNet*)
BNDatabaseGenerator is used to easily generate databases from a gum.BayesNet.

BNDatabaseGenerator(bn) -> BNDatabaseGenerator

Parameters:

- **bn** (*gum.BayesNet*) – the Bayesian network used to generate data.



database (*BNDatabaseGenerator self*)

drawSamples (*BNDatabaseGenerator self*, *int nbSamples*)

log2likelihood (*BNDatabaseGenerator self*)

setAntiTopologicalVarOrder (*BNDatabaseGenerator self*)

setRandomVarOrder (*BNDatabaseGenerator self*)

setTopologicalVarOrder (*BNDatabaseGenerator self*)

setVarOrder (*BNDatabaseGenerator self*, *vector< int, allocator< int > > varOrder*)
setVarOrder(BNDatabaseGenerator self, Vector_string varOrder)

setVarOrderFromCSV (*BNDatabaseGenerator self*, *str csvFileURL*, *str csvSeparator=', '*)

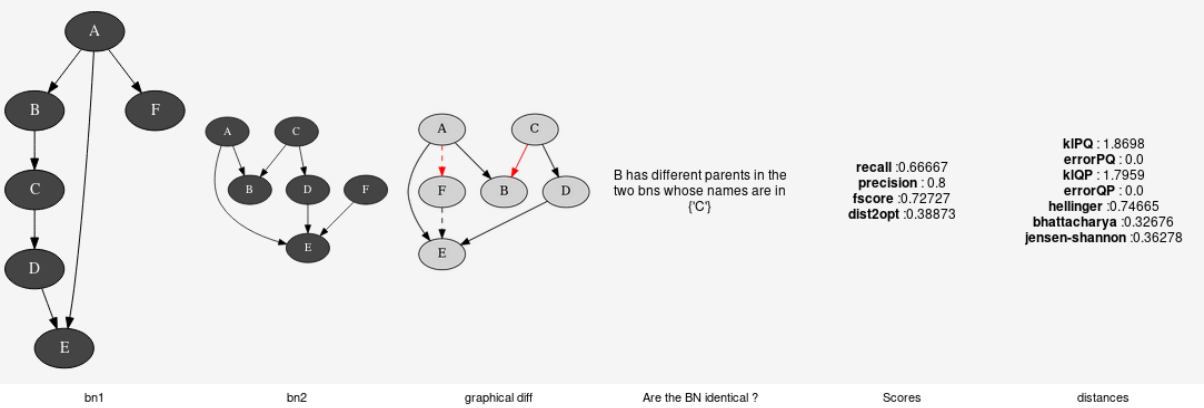
toCSV (*BNDatabaseGenerator self*, *str csvFileURL*, *bool useLabels=True*, *bool append=False*, *str csvSeparator=', '*, *bool checkOnAppend=False*)

toDatabaseTable (*BNDatabaseGenerator self*, *bool useLabels=True*)

varOrder (*BNDatabaseGenerator self*)

varOrderNames (*BNDatabaseGenerator self*)

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 `gum.OperationNotAllowed` – If the 2BNs have not the same domain size of compatible node sets

compute(*ExactBNdistance self*)

Returns a dictionnary containing the different values after the computation.

Return type dict

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 `gum.OperationNotAllowed` – If the 2BNs have not the same domain size of compatible node sets

burnIn(*GibbsBNdistance self*)

Returns size of burn in on number of iteration

Return type int

compute(*GibbsBNdistance self*)

Returns a dictionnary containing the different values after the computation.

Return type dict

continueApproximationScheme(*ApproximationScheme self*, *double error*)

Continue the approximation scheme.

Parameters **error** (*double*) –

currentTime(*GibbsBNdistance self*)

Returns get the current running time in second (double)

Return type double

disableEpsilon(*ApproximationScheme self*)

Disable epsilon as a stopping criterion.

disableMaxIter(*ApproximationScheme self*)

Disable max iterations as a stopping criterion.

disableMaxTime (*ApproximationScheme self*)
Disable max time as a stopping criterion.

disableMinEpsilonRate (*ApproximationScheme self*)
Disable a min epsilon rate as a stopping criterion.

enableEpsilon (*ApproximationScheme self*)
Enable epsilon as a stopping criterion.

enableMaxIter (*ApproximationScheme self*)
Enable max iterations as a stopping criterion.

enableMaxTime (*ApproximationScheme self*)
Enable max time as a stopping criterion.

enableMinEpsilonRate (*ApproximationScheme self*)
Enable a min epsilon rate as a stopping criterion.

epsilon (*GibbsBNdistance self*)

Returns the value of epsilon

Return type double

history (*GibbsBNdistance self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

initApproximationScheme (*ApproximationScheme self*)
Initiate the approximation scheme.

isDrawnAtRandom (*GibbsBNdistance self*)

Returns True if variables are drawn at random

Return type bool

isEnabledEpsilon (*ApproximationScheme self*)

Returns True if epsilon is used as a stopping criterion.

Return type bool

isEnabledMaxIter (*ApproximationScheme self*)

Returns True if max iterations is used as a stopping criterion

Return type bool

isEnabledMaxTime (*ApproximationScheme self*)

Returns True if max time is used as a stopping criterion

Return type bool

isEnabledMinEpsilonRate (*ApproximationScheme self*)

Returns True if epsilon rate is used as a stopping criterion

Return type bool

maxIter (*GibbsBNdistance self*)

Returns the criterion on number of iterations

Return type int

maxTime (*GibbsBNdistance self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*GibbsBNdistance self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*GibbsBNdistance self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*GibbsBNdistance self*)

Returns the number of variable drawn at each iteration

Return type int

nbrIterations(*GibbsBNdistance self*)

Returns the number of iterations

Return type int

periodSize(*GibbsBNdistance self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

remainingBurnIn(*ApproximationScheme self*)

Returns the number of remaining burn in

Return type int

setBurnIn(*GibbsBNdistance self*, *int b*)

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

setDrawnAtRandom(*GibbsBNdistance self*, *bool _atRandom*)

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

setEpsilon(*GibbsBNdistance self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

setMaxIter(*GibbsBNdistance self*, *int max*)

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

Raises `gum.OutOfLowerBound` – If max <= 1

setMaxTime(*GibbsBNdistance self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises `gum.OutOfLowerBound` – If timeout<=0.0

setMinEpsilonRate(*GibbsBNdistance self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setNbrDrawnVar(*GibbsBNdistance self*, *int _nbr*)

Parameters **_nbr** (*int*) – the number of variables to be drawn at each iteration

setPeriodSize(*GibbsBNdistance self*, *int p*)

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

Raises `gum.OutOfLowerBound` – If p<1

setVerbosity (*GibbsBNdistance self*, *bool v*)

Parameters **v** (*bool*) – verbosity

startOfPeriod (*ApproximationScheme self*)

Returns True if it is a start of a period

Return type bool

stateApproximationScheme (*ApproximationScheme self*)

Returns the state of the approximation scheme

Return type int

stopApproximationScheme (*ApproximationScheme self*)
Stop the approximation scheme.

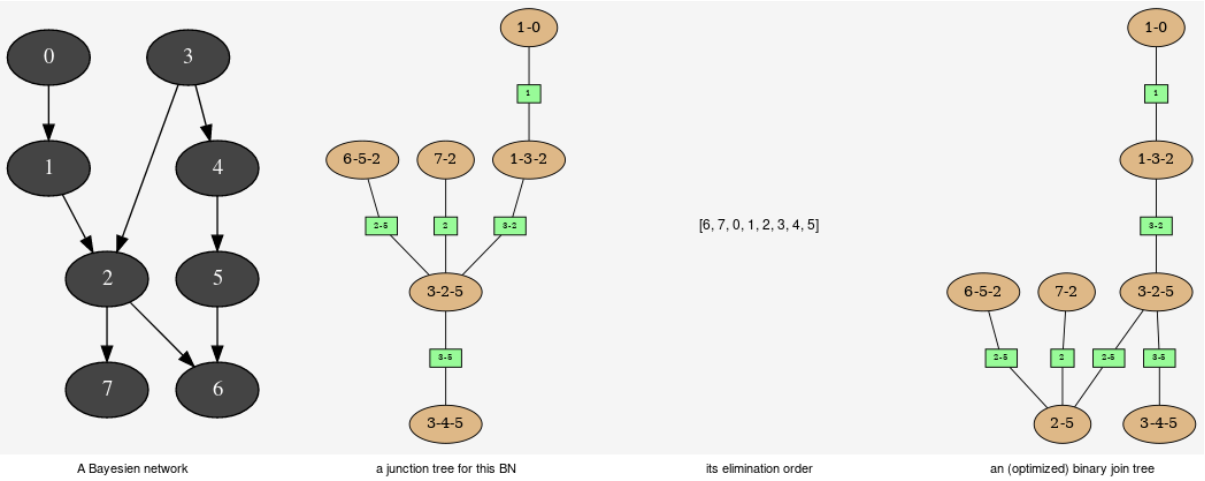
updateApproximationScheme (*ApproximationScheme self*, *unsigned int incr=1*)
Update the approximation scheme.

verbosity (*GibbsBNdistance self*)

Returns True if the verbosity is enabled

Return type bool

4.2.3 Explanation and analysis



This tools aimed to provide some different views on the Bayesian network in order to explore its qualitative and/or quantitave behaviours.

class `pyAgrum.`**JunctionTreeGenerator**
JunctionTreeGenerator is use to generate junction tree or binary junction tree from Bayesian networks.

JunctionTreeGenerator() -> JunctionTreeGenerator default constructor

binaryJoinTree (*JunctionTreeGenerator self*, *UndiGraph g*, *PyObject * partial_order=None*)
binaryJoinTree(JunctionTreeGenerator self, DAG dag, PyObject * partial_order=None) -> CliqueGraph binaryJoinTree(JunctionTreeGenerator self, BayesNet bn, PyObject * partial_order=None) -> CliqueGraph

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 9)) – a undirected graph
- **dag** (pyAgrum.DAG (page 7)) – a dag
- **bn** (pyAgrum.BayesNet (page 48)) – a BayesianNetwork
- **partial_order** (*List[List[int]]*) – a partial order among the nodeIDs

Returns the current binary joint tree

Return type *pyAgrum.CliqueGraph* (page 12)

eliminationOrder(*JunctionTreeGenerator self*, *UndiGraph g*, *PyObject * partial_order=None*)

eliminationOrder(JunctionTreeGenerator self, DAG dag, PyObject * partial_order=None) -> PyObject eliminationOrder(JunctionTreeGenerator self, BayesNet bn, PyObject * partial_order=None) -> PyObject

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 9)) – a undirected graph
- **dag** (pyAgrum.DAG (page 7)) – a dag
- **bn** (pyAgrum.BayesNet (page 48)) – a BayesianNetwork
- **partial_order** (*List[List[int]]*) – a partial order among the nodeIDs

Returns the current elimination order.

Return type *pyAgrum.CliqueGraph* (page 12)

junctionTree(*JunctionTreeGenerator self*, *UndiGraph g*, *PyObject * partial_order=None*)

junctionTree(JunctionTreeGenerator self, DAG dag, PyObject * partial_order=None) -> CliqueGraph junctionTree(JunctionTreeGenerator self, BayesNet bn, PyObject * partial_order=None) -> CliqueGraph junctionTree(JunctionTreeGenerator self, MarkovNet mn, PyObject * partial_order=None) -> CliqueGraph

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 9)) – a undirected graph
- **dag** (pyAgrum.DAG (page 7)) – a dag
- **bn** (pyAgrum.BayesNet (page 48)) – a BayesianNetwork
- **partial_order** (*List[List[int]]*) – a partial order among the nodeIDs

Returns the current junction tree.

Return type *pyAgrum.CliqueGraph* (page 12)

class pyAgrum.**EssentialGraph**(**args*)

Proxy of C++ pyAgrum.EssentialGraph class.

arcs(*EssentialGraph self*)

Returns The lisf of arcs in the EssentialGraph

Return type list

children(*EssentialGraph self*, *int 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(*EssentialGraph self*)

Returns the list of the edges

Return type List

mixedGraph(*EssentialGraph self*)

Returns the mixed graph

Return type *pyAgrum.MixedGraph* (page 16)

neighbours(*EssentialGraph self*, *int id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*EssentialGraph self*)

parents(*EssentialGraph self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

size(*EssentialGraph self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*EssentialGraph self*)

Returns the number of arcs in the graph

Return type int

sizeEdges(*EssentialGraph self*)

Returns the number of edges in the graph

Return type int

sizeNodes(*EssentialGraph self*)

Returns the number of nodes in the graph

Return type int

skeleton(*EssentialGraph self*)

toDot(*EssentialGraph self*)

Returns a friendly display of the graph in DOT format

Return type str

class pyAgrum.**MarkovBlanket**(**args*)

Proxy of C++ pyAgrum.MarkovBlanket class.

arcs(*MarkovBlanket self*)

Returns the list of the arcs

Return type List

children(*MarkovBlanket self*, *int 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(*MarkovBlanket self*)

Returns a copy of the DAG

Return type *pyAgrum.DAG* (page 7)

hasSameStructure(*MarkovBlanket self*, *DAGmodel other*)

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

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

Return type bool

nodes(*MarkovBlanket self*)

Returns the set of ids

Return type set

parents(*MarkovBlanket self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

size(*MarkovBlanket self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*MarkovBlanket self*)

Returns the number of arcs in the graph

Return type int

sizeNodes(*MarkovBlanket self*)

Returns the number of nodes in the graph

Return type int

toDot(*MarkovBlanket self*)

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: pyAgrum.IBayesNet*)

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

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(*BayesNetFragment self*, *PyObject * norid*)

arcs(*BayesNetFragment self*)

Returns The lisf of arcs in the IBayesNet

Return type list

checkConsistency(*BayesNetFragment self*, *int id*)

checkConsistency(BayesNetFragment self, str name) -> bool checkConsistency(BayesNetFragment self) -> bool

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 gum.NotFound – if the node is not found.

children(*BayesNetFragment self*, *PyObject * norid*)

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

Returns the set of all the children

Return type Set

completeInstantiation(*GraphicalModel self*)

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(*BayesNetFragment self*, *int varId*)
cpt(BayesNetFragment self, str name) -> Potential

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 39)

Raises gum.NotFound – If no variable's id matches varId.

dag(*BayesNetFragment self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

descendants(*BayesNetFragment self*, *PyObject * norid*)

dim(*IBayesNet self*)
Returns the dimension (the number of free parameters) in this BayesNet.

Returns the dimension of the BayesNet

Return type int

empty(*GraphicalModel self*)

exists(*DAGmodel self*, *int node*)

existsArc(*DAGmodel self*, *int tail*, *int head*)
existsArc(DAGmodel self, str nametail, str namehead) -> bool

family(*DAGmodel self*, *int id*)
family(DAGmodel self, str name) -> Set

hasSameStructure(*DAGmodel self*, *DAGmodel other*)

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

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

Return type bool

idFromName(*BayesNetFragment self*, *str 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 `gum.NotFound` – If name does not match a variable in the graph

ids(*GraphicalModel self*, *Vector_string names*)

installAscendants(*BayesNetFragment self*, *int id*)
installAscendants(BayesNetFragment self, str name)

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 `gum.NotFound` – if the node is not found.

installCPT(*BayesNetFragment self*, *int id*, *Potential pot*)
installCPT(BayesNetFragment self, str name, Potential pot)

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 39)) – the Potential to install

Raises `gum.NotFound` – if the node is not found.

installMarginal(*BayesNetFragment self*, *int id*, *Potential pot*)
installMarginal(BayesNetFragment self, str name, Potential pot)

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 39)) – the Potential (marginal) to install

Raises `gum.NotFound` – if the node is not found.

installNode(*BayesNetFragment self*, *int id*)
installNode(BayesNetFragment self, str name)

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 `gum.NotFound` – if the node is not found.

isIndependent(*BayesNetFragment self*, *PyObject * X*, *PyObject * Y*, *PyObject * Z*)

isInstalledNode(*BayesNetFragment self*, *int id*)
isInstalledNode(BayesNetFragment self, str name) -> bool

Check if a node is in the fragment

Parameters **n** (*int, str*) – the id or the name of the variable.

jointProbability(*IBayesNet self*, *Instantiation i*)

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

Returns a parameter of the joint probability for the BayesNet

Return type double

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

log10DomainSize(*GraphicalModel self*)

log2JointProbability(*IBayesNet self*, *Instantiation i*)

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

Returns a parameter of the log joint probability for the BayesNet

Return type double

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

maxNonOneParam(*IBayesNet self*)

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

Return type double

maxParam(*IBayesNet self*)

Returns the biggest value in the CPTs of the BayesNet

Return type double

maxVarDomainSize(*IBayesNet self*)

Returns the biggest domain size among the variables of the BayesNet

Return type int

minNonZeroParam(*IBayesNet self*)

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

Return type double

minParam(*IBayesNet self*)

Returns the smallest value in the CPTs of the IBayesNet

Return type double

minimalCondSet(*BayesNetFragment self*, *int target*, *PyObject * list*)

minimalCondSet(BayesNetFragment self, PyObject * targets, PyObject * list) -> PyObject *

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(*DAGmodel self*, *bool 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 9)

moralizedAncestralGraph(*BayesNetFragment self*, *PyObject * nodes*)

names(*BayesNetFragment self*)

Returns The names of the graph variables

Return type list

nodeId(*BayesNetFragment self*, *DiscreteVariable var*)

Parameters **var** (pyAgrum.DiscreteVariable (page 21)) – a variable

Returns the id of the variable

Return type int

Raises gum.IndexError – If the graph does not contain the variable

nodes(*BayesNetFragment self*)

Returns the set of ids

Return type set

nodeset(*GraphicalModel self*, *Vector_string names*)

parents(*BayesNetFragment self*, *PyObject * norid*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type Set

property(*GraphicalModel self*, *str name*)

propertyWithDefault(*GraphicalModel self*, *str name*, *str byDefault*)

setProperty(*GraphicalModel self*, *str name*, *str value*)

size(*DAGmodel self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*DAGmodel self*)

Returns the number of arcs in the graph

Return type int

toBN(*BayesNetFragment self*)
Create a BayesNet from a fragment.

Raises gum.OperationNotAllowed – if the fragment is not consistent.

toDot(*BayesNetFragment self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*DAGmodel self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises gum.InvalidDirectedCycle – If this graph contains cycles

uninstallCPT(*BayesNetFragment self*, *int id*)
uninstallCPT(BayesNetFragment self, str name)

Remove a local CPT. The fragment can become inconsistant.

Parameters **n** (*int, str*) – the id or the name of the variable.

Raises gum.NotFound – if the node is not found.

uninstallNode(*BayesNetFragment self*, *int id*)
uninstallNode(BayesNetFragment self, str name)

Remove a node from the fragment. The fragment can become inconsistant.

Parameters **n** (*int, str*) – the id or the name of the variable.

Raises gum.NotFound – if the node is not found.

variable(*BayesNetFragment self*, *int id*)
variable(BayesNetFragment self, str name) -> DiscreteVariable

Parameters

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises gum.IndexError – If the graph does not contain the variable

variableFromName(*BayesNetFragment self*, *str name*)

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises gum.IndexError – If the graph does not contain the variable

variableNodeMap(*BayesNetFragment self*)

Returns the variable node map

Return type pyAgrum.variableNodeMap

whenArcAdded(*BayesNetFragment self*, *void * src*, *int _from*, *int to*)

whenArcDeleted(*BayesNetFragment self*, *void * src*, *int _from*, *int to*)

whenNodeAdded(*BayesNetFragment self*, *void * src*, *int id*)

whenNodeDeleted(*BayesNetFragment self*, *void * src*, *int id*)

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(*LazyPropagation self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises `gum.UndefinedElement` – If no Bayes net has been assigned to the inference.

H(*LazyPropagation self*, *int X*)
H(LazyPropagation self, str nodeName) -> double

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 double

I(*LazyPropagation self*, *int X*, *int Y*)
I(LazyPropagation self, str X, str Y) -> double

Parameters

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

Returns

- **-------** –
- **double** – the Mutual Information of X and Y given the observation

VI(*LazyPropagation self*, *int X*, *int Y*)
VI(LazyPropagation self, str X, str Y) -> double

Parameters

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

Returns

- **-------** –
- **double** – variation of information between X and Y

addAllTargets(*LazyPropagation self*)
Add all the nodes as targets.

addEvidence(*LazyPropagation self*, *int id*, *int val*)
addEvidence(LazyPropagation self, str nodeName, int val) addEvidence(LazyPropagation self, int id, str val) addEvidence(LazyPropagation self, str nodeName, str val) addEvidence(LazyPropagation self, int id, Vector vals) addEvidence(LazyPropagation self, str nodeName, Vector vals)

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

- `gum.InvalidArgument` – If the node already has an evidence

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

addJointTarget (*LazyPropagation self*, *PyObject * 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

Raises gum.UndefinedElement – If some node(s) do not belong to the Bayesian network

addTarget (*LazyPropagation self*, *int target*)

addTarget(LazyPropagation self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence (*LazyPropagation self*, *int id*, *int val*)

chgEvidence(LazyPropagation self, str nodeName, int val) chgEvidence(LazyPropagation self, int id, str val) chgEvidence(LazyPropagation self, str nodeName, str val) chgEvidence(LazyPropagation self, int id, Vector vals) chgEvidence(LazyPropagation self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

eraseAllEvidence (*LazyPropagation self*)

Removes all the evidence entered into the network.

eraseAllJointTargets (*LazyPropagation self*)

Clear all previously defined joint targets.

eraseAllMarginalTargets (*LazyPropagation self*)

Clear all the previously defined marginal targets.

eraseAllTargets (*LazyPropagation self*)
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).

eraseEvidence (*LazyPropagation self*, *int id*)
eraseEvidence(LazyPropagation self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises gum.IndexError – If the node does not belong to the Bayesian network

eraseJointTarget (*LazyPropagation self*, *PyObject * targets*)
Remove, if existing, the joint target.

Parameters **list** – a list of names or Ids of nodes

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

eraseTarget (*LazyPropagation self*, *int target*)
eraseTarget(LazyPropagation self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

evidenceImpact (*LazyPropagation self*, *PyObject * target*, *PyObject * 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 39)

evidenceJointImpact (*LazyPropagation self*, *PyObject * targets*, *PyObject * evs*)
evidenceJointImpact(LazyPropagation self, Vector_string targets, Vector_string evs) -> Potential

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(targetlevs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceProbability (*LazyPropagation self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes (*LazyPropagation self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*LazyPropagation self*, *int id*)
hasEvidence(LazyPropagation self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence (*LazyPropagation self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence (*LazyPropagation self*, *int id*)
hasSoftEvidence(LazyPropagation self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

isJointTarget (*LazyPropagation self*, *PyObject * targets*)

Parameters **list** – a list of nodes ids or names.

Returns True if target is a joint target.

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

isTarget(*LazyPropagation self*, *int variable*)
isTarget(LazyPropagation self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

joinTree(*LazyPropagation self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*LazyPropagation self*, *PyObject * targets*)

jointPosterior(*LazyPropagation self*, *PyObject * 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 ref to the posterior joint probability of the set of nodes.

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

jointTargets(*LazyPropagation self*)

Returns the list of target sets

Return type list

junctionTree(*LazyPropagation self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*LazyPropagation self*)
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.

nbrEvidence(*LazyPropagation self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LazyPropagation self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*LazyPropagation self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*LazyPropagation self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LazyPropagation self*)

Returns the number of marginal targets

Return type int

posterior(*LazyPropagation self*, *int var*)

posterior(LazyPropagation self, str nodeName) -> Potential posterior(LazyPropagation self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – 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

setFindBarrenNodesType(*LazyPropagation self*, *pyAgrum.FindBarrenNodesType type*)

sets how we determine barren nodes

Barren nodes are unnecessary for probability inference, so they can be safely discarded in this case (type = FIND_BARREN_NODES). This speeds-up inference. However, there are some cases in which we do not want to remove barren nodes, typically when we want to answer queries such as Most Probable Explanations (MPE).

0 = FIND_NO_BARREN_NODES 1 = FIND_BARREN_NODES

Parameters **type** (*int*) – the finder type

Raises gum.InvalidArgument – If type is not implemented

setRelevantPotentialsFinderType(*LazyPropagation self*, *pyAgrum.RelevantPotentialsFinderType type*)

sets how we determine the relevant potentials to combine

When a clique sends a message to a separator, it first constitute the set of the potentials it contains and of the potentials contained in the messages it received. If RelevantPotentialsFinderType = FIND_ALL, all these potentials are combined and projected to produce the message sent to the separator. If RelevantPotentialsFinderType = DSEP_BAYESBALL_NODES, then only the set of potentials d-connected to the variables of the separator are kept for combination and projection.

0 = FIND_ALL 1 = DSEP_BAYESBALL_NODES 2 = DSEP_BAYESBALL_POTENTIALS 3 = DSEP_KOLLER_FRIEDMAN_2009

Parameters **type** (*int*) – the finder type

Raises gum.InvalidArgument – If type is not implemented

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

setTriangulation(*LazyPropagation self*, *Triangulation new_triangulation*)

softEvidenceNodes(*LazyPropagation self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LazyPropagation self*)

Returns the list of marginal targets

Return type list

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(*ShaferShenoyInference self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises `gum.UndefinedElement` – If no Bayes net has been assigned to the inference.

H(*ShaferShenoyInference self*, *int X*)
H(ShaferShenoyInference self, str nodeName) -> double

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 double

I(*ShaferShenoyInference self*, *int X*, *int Y*)
I(ShaferShenoyInference self, str X, str Y) -> double

Parameters

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

Returns

- **-------** –
- **double** – the Mutual Information of X and Y given the observation

VI(*ShaferShenoyInference self*, *int X*, *int Y*)
VI(ShaferShenoyInference self, str X, str Y) -> double

Parameters

- **X** (*int or str*) – a node Id or a node name
- **Y** (*int or str*) – another node Id or node name

Returns

- **-------** –
- **double** – variation of information between X and Y

addAllTargets(*ShaferShenoyInference self*)
Add all the nodes as targets.

addEvidence(*ShaferShenoyInference self*, *int id*, *int val*)
addEvidence(ShaferShenoyInference self, str nodeName, int val) addEvidence(ShaferShenoyInference self, int id, str val) addEvidence(ShaferShenoyInference self, str nodeName, str val) addEvidence(ShaferShenoyInference self, int id, Vector vals) addEvidence(ShaferShenoyInference self, str nodeName, Vector vals)

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

- `gum.InvalidArgument` – If the node already has an evidence
- `gum.InvalidArgument` – If val is not a value for the node

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

addJointTarget(*ShaferShenoyInference self*, *PyObject * 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

Raises gum.UndefinedElement – If some node(s) do not belong to the Bayesian network

addTarget(*ShaferShenoyInference self*, *int target*)

addTarget(ShaferShenoyInference self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence(*ShaferShenoyInference self*, *int id*, *int val*)

chgEvidence(ShaferShenoyInference self, str nodeName, int val) chgEvidence(ShaferShenoyInference self, int id, str val) chgEvidence(ShaferShenoyInference self, str nodeName, str val) chgEvidence(ShaferShenoyInference self, int id, Vector vals) chgEvidence(ShaferShenoyInference self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

eraseAllEvidence(*ShaferShenoyInference self*)

Removes all the evidence entered into the network.

eraseAllJointTargets(*ShaferShenoyInference self*)

Clear all previously defined joint targets.

eraseAllMarginalTargets(*ShaferShenoyInference self*)

Clear all the previously defined marginal targets.

eraseAllTargets (*ShaferShenoyInference self*)
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).

eraseEvidence (*ShaferShenoyInference self*, *int id*)
eraseEvidence(ShaferShenoyInference self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseJointTarget (*ShaferShenoyInference self*, *PyObject * targets*)
Remove, if existing, the joint target.

Parameters **list** – a list of names or Ids of nodes

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

eraseTarget (*ShaferShenoyInference self*, *int target*)
eraseTarget(ShaferShenoyInference self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact (*ShaferShenoyInference self*, *PyObject * target*, *PyObject * 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 39)

evidenceJointImpact (*ShaferShenoyInference self*, *PyObject * targets*, *PyObject * evs*)
evidenceJointImpact(ShaferShenoyInference self, Vector_string targets, Vector_string evs) -> Potential

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(targetlevs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceProbability(*ShaferShenoyInference self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes(*ShaferShenoyInference self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*ShaferShenoyInference self*, *int id*)
hasEvidence(ShaferShenoyInference self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence(*ShaferShenoyInference self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*ShaferShenoyInference self*, *int id*)
hasSoftEvidence(ShaferShenoyInference self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

isJointTarget(*ShaferShenoyInference self*, *PyObject * targets*)

Parameters **list** – a list of nodes ids or names.

Returns True if target is a joint target.

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

isTarget(*ShaferShenoyInference self*, *int variable*)
isTarget(ShaferShenoyInference self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

joinTree(*ShaferShenoyInference self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*ShaferShenoyInference self*, *PyObject * targets*)

jointPosterior(*ShaferShenoyInference self*, *PyObject * 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 ref to the posterior joint probability of the set of nodes.

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

jointTargets(*ShaferShenoyInference self*)

Returns the list of target sets

Return type list

junctionTree(*ShaferShenoyInference self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*ShaferShenoyInference self*)
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.

nbrEvidence(*ShaferShenoyInference self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*ShaferShenoyInference self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*ShaferShenoyInference self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*ShaferShenoyInference self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*ShaferShenoyInference self*)

Returns the number of marginal targets

Return type int

posterior(*ShaferShenoyInference self*, *int var*)

posterior(ShaferShenoyInference self, str nodeName) -> Potential posterior(ShaferShenoyInference self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – 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

setFindBarrenNodesType(*ShaferShenoyInference self*, *pyAgrum.FindBarrenNodesType type*)

sets how we determine barren nodes

Barren nodes are unnecessary for probability inference, so they can be safely discarded in this case (type = FIND_BARREN_NODES). This speeds-up inference. However, there are some cases in which we do not want to remove barren nodes, typically when we want to answer queries such as Most Probable Explanations (MPE).

0 = FIND_NO_BARREN_NODES 1 = FIND_BARREN_NODES

Parameters **type** (*int*) – the finder type

Raises `gum.InvalidArgument` – If type is not implemented

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

setTriangulation(*ShaferShenoyInference self*, *Triangulation new_triangulation*)

softEvidenceNodes(*ShaferShenoyInference self*)

Returns the set of nodes with soft evidence

Return type set

targets(*ShaferShenoyInference self*)

Returns the list of marginal targets

Return type list

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(*VariableElimination self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*VariableElimination self*, *int X*)

H(VariableElimination self, str nodeName) -> double

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 double

addAllTargets(*VariableElimination self*)

Add all the nodes as targets.

addEvidence(*VariableElimination self*, *int id*, *int val*)
addEvidence(VariableElimination self, str nodeName, int val) addEvidence(VariableElimination self, int id, str val) addEvidence(VariableElimination self, str nodeName, str val) addEvidence(VariableElimination self, int id, Vector vals) addEvidence(VariableElimination self, str nodeName, Vector vals)

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

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

addJointTarget(*VariableElimination self*, *PyObject * 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

Raises `gum.UndefinedElement` – If some node(s) do not belong to the Bayesian network

addTarget(*VariableElimination self*, *int target*)
addTarget(VariableElimination self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

chgEvidence(*VariableElimination self*, *int id*, *int val*)
chgEvidence(VariableElimination self, str nodeName, int val) chgEvidence(VariableElimination self, int id, str val) chgEvidence(VariableElimination self, str nodeName, str val) chgEvidence(VariableElimination self, int id, Vector vals) chgEvidence(VariableElimination self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

eraseAllEvidence(*VariableElimination self*)
Removes all the evidence entered into the network.

eraseAllTargets(*VariableElimination self*)
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).

eraseEvidence(*VariableElimination self*, *int id*)
eraseEvidence(VariableElimination self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises gum.IndexError – If the node does not belong to the Bayesian network

eraseJointTarget(*VariableElimination self*, *PyObject * targets*)
Remove, if existing, the joint target.

Parameters **list** – a list of names or Ids of nodes

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

eraseTarget(*VariableElimination self*, *int target*)
eraseTarget(VariableElimination self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

evidenceImpact(*VariableElimination self*, *PyObject * target*, *PyObject * evs*)
Create a pyAgrum.Potential for P(targetlevs) (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 39)

evidenceJointImpact (*VariableElimination self*, *PyObject * targets*, *PyObject * evs*)
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 39)

Raises `gum.Exception` – If some evidene entered into the Bayes net are incompatible (their joint proba = 0)

hardEvidenceNodes (*VariableElimination self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*VariableElimination self*, *int id*)
hasEvidence(VariableElimination self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence (*VariableElimination self*, *str 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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasSoftEvidence (*VariableElimination self*, *int id*)
hasSoftEvidence(VariableElimination self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

isJointTarget(*VariableElimination self*, *PyObject * targets*)

Parameters **list** – a list of nodes ids or names.

Returns True if target is a joint target.

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

isTarget(*VariableElimination self*, *int variable*)
isTarget(VariableElimination self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

jointMutualInformation(*VariableElimination self*, *PyObject * targets*)

jointPosterior(*VariableElimination self*, *PyObject * 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 ref to the posterior joint probability of the set of nodes.

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – If an element of nodes is not in targets

jointTargets(*VariableElimination self*)

Returns the list of target sets

Return type list

junctionTree(*VariableElimination self*, *int id*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*VariableElimination self*)
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.

nbrEvidence(*VariableElimination self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*VariableElimination self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrSoftEvidence(*VariableElimination self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*VariableElimination self*)

Returns the number of marginal targets

Return type int

posterior(*VariableElimination self*, *int var*)
posterior(VariableElimination self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – 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

setFindBarrenNodesType(*VariableElimination self*, *pyAgrum.FindBarrenNodesType type*)
sets how we determine barren nodes

Barren nodes are unnecessary for probability inference, so they can be safely discarded in this case (type = FIND_BARREN_NODES). This speeds-up inference. However, there are some cases in which we do not want to remove barren nodes, typically when we want to answer queries such as Most Probable Explanations (MPE).

0 = FIND_NO_BARREN_NODES 1 = FIND_BARREN_NODES

Parameters **type** (*int*) – the finder type

Raises gum.InvalidArgument – If type is not implemented

setRelevantPotentialsFinderType(*VariableElimination self*, *pyAgrum.RelevantPotentialsFinderType type*)
sets how we determine the relevant potentials to combine

When a clique sends a message to a separator, it first constitute the set of the potentials it contains and of the potentials contained in the messages it received. If RelevantPotentialsFinderType =

FIND_ALL, all these potentials are combined and projected to produce the message sent to the separator. If RelevantPotentialsFinderType = DSEP_BAYESBALL_NODES, then only the set of potentials d-connected to the variables of the separator are kept for combination and projection.

0 = FIND_ALL 1 = DSEP_BAYESBALL_NODES 2 = DSEP_BAYESBALL_POTENTIALS 3 = DSEP_KOLLER_FRIEDMAN_2009

Parameters **type** (*int*) – the finder type

Raises gum.InvalidArgument – If type is not implemented

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

setTriangulation(*VariableElimination self*, *Triangulation new_triangulation*)

softEvidenceNodes(*VariableElimination self*)

Returns the set of nodes with soft evidence

Return type set

targets(*VariableElimination self*)

Returns the list of marginal targets

Return type list

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: pyAgrum.IBayesNet*)
Class used for inferences using loopy belief propagation algorithm.

LoopyBeliefPropagation(bn) -> LoopyBeliefPropagation

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*LoopyBeliefPropagation self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*LoopyBeliefPropagation self*, *int X*)
H(LoopyBeliefPropagation self, str nodeName) -> double

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 double

addAllTargets(*LoopyBeliefPropagation self*)
Add all the nodes as targets.

addEvidence(*LoopyBeliefPropagation self*, *int id*, *int val*)
addEvidence(LoopyBeliefPropagation self, str nodeName, int val) addEvidence(LoopyBeliefPropagation self, int id, str val) addEvidence(LoopyBeliefPropagation self, str nodeName, str val) addEvidence(LoopyBeliefPropagation self, int id, Vector vals) addEvidence(LoopyBeliefPropagation self, str nodeName, Vector vals)

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

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

addTarget(*LoopyBeliefPropagation self*, *int target*)
addTarget(LoopyBeliefPropagation self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

chgEvidence(*LoopyBeliefPropagation self*, *int id*, *int val*)
chgEvidence(LoopyBeliefPropagation self, str nodeName, int val) chgEvidence(LoopyBeliefPropagation self, int id, str val) chgEvidence(LoopyBeliefPropagation self, str nodeName, str val) chgEvidence(LoopyBeliefPropagation self, int id, Vector vals) chgEvidence(LoopyBeliefPropagation self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentTime(*LoopyBeliefPropagation self*)

Returns get the current running time in second (double)

Return type double

epsilon(*LoopyBeliefPropagation self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*LoopyBeliefPropagation self*)
Removes all the evidence entered into the network.

eraseAllTargets(*LoopyBeliefPropagation self*)
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).

eraseEvidence(*LoopyBeliefPropagation self*, *int id*)
eraseEvidence(LoopyBeliefPropagation self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises gum.IndexError – If the node does not belong to the Bayesian network

eraseTarget(*LoopyBeliefPropagation self*, *int target*)
eraseTarget(LoopyBeliefPropagation self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

evidenceImpact(*LoopyBeliefPropagation self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes(*LoopyBeliefPropagation self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*LoopyBeliefPropagation self*, *int id*)
hasEvidence(LoopyBeliefPropagation self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*LoopyBeliefPropagation self*, *str 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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasSoftEvidence(*LoopyBeliefPropagation self*, *int id*)
hasSoftEvidence(LoopyBeliefPropagation self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

history(*LoopyBeliefPropagation self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

isTarget(*LoopyBeliefPropagation self*, *int variable*)
isTarget(LoopyBeliefPropagation self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

makeInference(*LoopyBeliefPropagation self*)
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.

maxIter(*LoopyBeliefPropagation self*)

Returns the criterion on number of iterations

Return type int

maxTime(*LoopyBeliefPropagation self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*LoopyBeliefPropagation self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*LoopyBeliefPropagation self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*LoopyBeliefPropagation self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LoopyBeliefPropagation self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*LoopyBeliefPropagation self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*LoopyBeliefPropagation self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LoopyBeliefPropagation self*)

Returns the number of marginal targets

Return type int

periodSize(*LoopyBeliefPropagation self*)

Returns the number of samples between 2 stopping

Return type int

Raises gum.OutOfLowerBound – If p<1

posterior(*LoopyBeliefPropagation self*, *int var*)
posterior(LoopyBeliefPropagation self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – If an element of nodes is not in targets

setEpsilon(*LoopyBeliefPropagation self*, *double eps*)

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

Raises gum.OutOfLowerBound – If eps<0

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(*LoopyBeliefPropagation self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*LoopyBeliefPropagation self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*LoopyBeliefPropagation self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*LoopyBeliefPropagation self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*LoopyBeliefPropagation self*, *bool v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*LoopyBeliefPropagation self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LoopyBeliefPropagation self*)

Returns the list of marginal targets

Return type list

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(*LoopyBeliefPropagation self*)

Returns True if the verbosity is enabled

Return type bool

4.5.2 Sampling

Gibbs Sampling

class pyAgrum.**GibbsSampling**(*bn: pyAgrum.IBayesNet*)
Class for making Gibbs sampling inference in Bayesian networks.

GibbsSampling(bn) -> GibbsSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*GibbsSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*GibbsSampling self*, *int X*)
H(GibbsSampling self, str nodeName) -> double

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 double

addAllTargets (*GibbsSampling self*)
Add all the nodes as targets.

addEvidence (*GibbsSampling self*, *int id*, *int val*)
addEvidence(GibbsSampling self, str nodeName, int val) addEvidence(GibbsSampling self, int id, str val) addEvidence(GibbsSampling self, str nodeName, str val) addEvidence(GibbsSampling self, int id, Vector vals) addEvidence(GibbsSampling self, str nodeName, Vector vals)

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

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

addTarget (*GibbsSampling self*, *int target*)
addTarget(GibbsSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

burnIn (*GibbsSampling self*)

Returns size of burn in on number of iteration

Return type int

chgEvidence (*GibbsSampling self*, *int id*, *int val*)
chgEvidence(GibbsSampling self, str nodeName, int val) chgEvidence(GibbsSampling self, int id, str val) chgEvidence(GibbsSampling self, str nodeName, str val) chgEvidence(GibbsSampling self, int id, Vector vals) chgEvidence(GibbsSampling self, str nodeName, Vector vals)

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

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

currentPosterior(*GibbsSampling self*, *int id*)
currentPosterior(GibbsSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*GibbsSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*GibbsSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*GibbsSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets(*GibbsSampling self*)
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).

eraseEvidence(*GibbsSampling self*, *int id*)
eraseEvidence(GibbsSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget(*GibbsSampling self*, *int target*)
eraseTarget(GibbsSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact(*GibbsSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes(*GibbsSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*GibbsSampling self*, *int id*)
hasEvidence(GibbsSampling self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*GibbsSampling self*, *str 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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasSoftEvidence(*GibbsSampling self*, *int id*)
hasSoftEvidence(GibbsSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

history(*GibbsSampling self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

isDrawnAtRandom(*GibbsSampling self*)

Returns True if variables are drawn at random

Return type bool

isTarget(*GibbsSampling self*, *int variable*)
isTarget(GibbsSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

makeInference(*GibbsSampling self*)
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.

maxIter(*GibbsSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*GibbsSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*GibbsSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*GibbsSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*GibbsSampling self*)

Returns the number of variable drawn at each iteration

Return type int

nbrEvidence(*GibbsSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*GibbsSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*GibbsSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*GibbsSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*GibbsSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*GibbsSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*GibbsSampling self*, *int var*)
posterior(GibbsSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setBurnIn(*GibbsSampling self*, *int b*)

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

setDrawnAtRandom(*GibbsSampling self*, *bool _atRandom*)

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

setEpsilon(*GibbsSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*GibbsSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*GibbsSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*GibbsSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setNbrDrawnVar(*GibbsSampling self*, *int _nbr*)

Parameters **_nbr** (*int*) – the number of variables to be drawn at each iteration

setPeriodSize(*GibbsSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*GibbsSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*GibbsSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*GibbsSampling self*)

Returns the list of marginal targets

Return type list

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(*GibbsSampling self*)

Returns True if the verbosity is enabled

Return type bool

Monte Carlo Sampling

class pyAgrum.**MonteCarloSampling**(*bn: pyAgrum.IBayesNet*)
Class used for Monte Carlo sampling inference algorithm.

MonteCarloSampling(bn) -> MonteCarloSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*MonteCarloSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*MonteCarloSampling self*, *int X*)
H(MonteCarloSampling self, str nodeName) -> double

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 double

addAllTargets(*MonteCarloSampling self*)
Add all the nodes as targets.

addEvidence(*MonteCarloSampling self*, *int id*, *int val*)
addEvidence(MonteCarloSampling self, str nodeName, int val) addEvidence(MonteCarloSampling self, int id, str val) addEvidence(MonteCarloSampling self, str nodeName, str val) addEvidence(MonteCarloSampling self, int id, Vector vals) addEvidence(MonteCarloSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node already has an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

addTarget(*MonteCarloSampling self*, *int target*)
addTarget(MonteCarloSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence(*MonteCarloSampling self*, *int id*, *int val*)
chgEvidence(MonteCarloSampling self, str nodeName, int val) chgEvidence(MonteCarloSampling self, int id, str val) chgEvidence(MonteCarloSampling self, str nodeName, str val) chgEvidence(MonteCarloSampling self, int id, Vector vals) chgEvidence(MonteCarloSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior(*MonteCarloSampling self*, *int id*)
currentPosterior(MonteCarloSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*MonteCarloSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*MonteCarloSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*MonteCarloSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets (*MonteCarloSampling self*)
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).

eraseEvidence (*MonteCarloSampling self*, *int id*)
eraseEvidence(MonteCarloSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget (*MonteCarloSampling self*, *int target*)
eraseTarget(MonteCarloSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact (*MonteCarloSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes (*MonteCarloSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*MonteCarloSampling self*, *int id*)
hasEvidence(MonteCarloSampling self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*MonteCarloSampling self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*MonteCarloSampling self*, *int id*)
hasSoftEvidence(MonteCarloSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

history(*MonteCarloSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

isTarget(*MonteCarloSampling self*, *int variable*)
isTarget(MonteCarloSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

makeInference(*MonteCarloSampling self*)
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.

maxIter(*MonteCarloSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*MonteCarloSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*MonteCarloSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*MonteCarloSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*MonteCarloSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*MonteCarloSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*MonteCarloSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*MonteCarloSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*MonteCarloSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*MonteCarloSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*MonteCarloSampling self*, *int var*)

posterior(MonteCarloSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon(*MonteCarloSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*MonteCarloSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*MonteCarloSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*MonteCarloSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*MonteCarloSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*MonteCarloSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*MonteCarloSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*MonteCarloSampling self*)

Returns the list of marginal targets

Return type list

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(*MonteCarloSampling self*)

Returns True if the verbosity is enabled

Return type bool

Weighted Sampling

class pyAgrum.**WeightedSampling**(*bn: pyAgrum.IBayesNet*)
Class used for Weighted sampling inference algorithm.

WeightedSampling(bn) -> WeightedSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*WeightedSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*WeightedSampling self*, *int X*)
H(WeightedSampling self, str nodeName) -> double

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 double

addAllTargets(*WeightedSampling self*)
Add all the nodes as targets.

addEvidence(*WeightedSampling self*, *int id*, *int val*)
addEvidence(WeightedSampling self, str nodeName, int val) addEvidence(WeightedSampling self, int id, str val) addEvidence(WeightedSampling self, str nodeName, str val) addEvidence(WeightedSampling self, int id, Vector vals) addEvidence(WeightedSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node already has an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

addTarget(*WeightedSampling self*, *int target*)
addTarget(WeightedSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence(*WeightedSampling self*, *int id*, *int val*)
chgEvidence(WeightedSampling self, str nodeName, int val) chgEvidence(WeightedSampling self, int id, str val) chgEvidence(WeightedSampling self, str nodeName, str val) chgEvidence(WeightedSampling self, int id, Vector vals) chgEvidence(WeightedSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior(*WeightedSampling self*, *int id*)
currentPosterior(WeightedSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*WeightedSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*WeightedSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence (*WeightedSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets (*WeightedSampling self*)
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).

eraseEvidence (*WeightedSampling self*, *int id*)
eraseEvidence(WeightedSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget (*WeightedSampling self*, *int target*)
eraseTarget(WeightedSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact (*WeightedSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes (*WeightedSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*WeightedSampling self*, *int id*)
hasEvidence(WeightedSampling self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence(*WeightedSampling self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*WeightedSampling self*, *int id*)
hasSoftEvidence(WeightedSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

history(*WeightedSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

isTarget(*WeightedSampling self*, *int variable*)
isTarget(WeightedSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

makeInference(*WeightedSampling self*)
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.

maxIter(*WeightedSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*WeightedSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*WeightedSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*WeightedSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*WeightedSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*WeightedSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*WeightedSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*WeightedSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*WeightedSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*WeightedSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*WeightedSampling self*, *int var*)

posterior(WeightedSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon(*WeightedSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*WeightedSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*WeightedSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*WeightedSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*WeightedSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*WeightedSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*WeightedSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*WeightedSampling self*)

Returns the list of marginal targets

Return type list

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 (*WeightedSampling self*)

Returns True if the verbosity is enabled

Return type bool

Importance Sampling

class pyAgrum.**ImportanceSampling**(*bn: pyAgrum.IBayesNet*)
Class used for inferences using the Importance Sampling algorithm.

ImportanceSampling(bn) -> ImportanceSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*ImportanceSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises `gum.UndefinedElement` – If no Bayes net has been assigned to the inference.

H(*ImportanceSampling self*, *int X*)
H(ImportanceSampling self, str nodeName) -> double

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 double

addAllTargets(*ImportanceSampling self*)
Add all the nodes as targets.

addEvidence(*ImportanceSampling self*, *int id*, *int val*)
addEvidence(ImportanceSampling self, str nodeName, int val) addEvidence(ImportanceSampling self, int id, str val) addEvidence(ImportanceSampling self, str nodeName, str val) addEvidence(ImportanceSampling self, int id, Vector vals) addEvidence(ImportanceSampling self, str nodeName, Vector vals)

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

- `gum.InvalidArgument` – If the node already has an evidence
- `gum.InvalidArgument` – If val is not a value for the node
- `gum.InvalidArgument` – If the size of vals is different from the domain side of the node
- `gum.FatalError` – If vals is a vector of 0s

- gum.UndefinedElement – If the node does not belong to the Bayesian network

addTarget(*ImportanceSampling self*, *int target*)
addTarget(ImportanceSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence(*ImportanceSampling self*, *int id*, *int val*)
chgEvidence(ImportanceSampling self, str nodeName, int val) chgEvidence(ImportanceSampling self, int id, str val) chgEvidence(ImportanceSampling self, str nodeName, str val) chgEvidence(ImportanceSampling self, int id, Vector vals) chgEvidence(ImportanceSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior(*ImportanceSampling self*, *int id*)
currentPosterior(ImportanceSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*ImportanceSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*ImportanceSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*ImportanceSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets(*ImportanceSampling self*)
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).

eraseEvidence(*ImportanceSampling self*, *int id*)
eraseEvidence(ImportanceSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget(*ImportanceSampling self*, *int target*)
eraseTarget(ImportanceSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact(*ImportanceSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes(*ImportanceSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*ImportanceSampling self*, *int id*)
hasEvidence(ImportanceSampling self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*ImportanceSampling self*, *str 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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasSoftEvidence(*ImportanceSampling self*, *int id*)
hasSoftEvidence(ImportanceSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

history(*ImportanceSampling self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

isTarget(*ImportanceSampling self*, *int variable*)
isTarget(ImportanceSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

makeInference(*ImportanceSampling self*)
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.

maxIter(*ImportanceSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime (*ImportanceSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme (*ImportanceSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate (*ImportanceSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence (*ImportanceSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence (*ImportanceSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations (*ImportanceSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence (*ImportanceSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets (*ImportanceSampling self*)

Returns the number of marginal targets

Return type int

periodSize (*ImportanceSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior (*ImportanceSampling self*, *int var*)

posterior(ImportanceSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon (*ImportanceSampling self*, *double eps*)

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

Raises gum.OutOfLowerBound – If eps<0

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(*ImportanceSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*ImportanceSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*ImportanceSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*ImportanceSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*ImportanceSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*ImportanceSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*ImportanceSampling self*)

Returns the list of marginal targets

Return type list

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 (*ImportanceSampling self*)

Returns True if the verbosity is enabled

Return type bool

4.5.3 Loopy sampling

Loopy Gibbs Sampling

class pyAgrum.**LoopyGibbsSampling**(*bn: pyAgrum.IBayesNet*)

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

LoopyGibbsSampling(bn) -> LoopyGibbsSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*LoopyGibbsSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*LoopyGibbsSampling self*, *int X*)

H(LoopyGibbsSampling self, str nodeName) -> double

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 double

addAllTargets(*LoopyGibbsSampling self*)

Add all the nodes as targets.

addEvidence(*LoopyGibbsSampling self*, *int id*, *int val*)

addEvidence(LoopyGibbsSampling self, str nodeName, int val) addEvidence(LoopyGibbsSampling self, int id, str val) addEvidence(LoopyGibbsSampling self, str nodeName, str val) addEvidence(LoopyGibbsSampling self, int id, Vector vals) addEvidence(LoopyGibbsSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node already has an evidence
- gum.InvalidArgument – If val is not a value for the node

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

addTarget (*LoopyGibbsSampling self*, *int target*)
addTarget(LoopyGibbsSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

burnIn (*LoopyGibbsSampling self*)

Returns size of burn in on number of iteration

Return type int

chgEvidence (*LoopyGibbsSampling self*, *int id*, *int val*)
chgEvidence(LoopyGibbsSampling self, str nodeName, int val) chgEvidence(LoopyGibbsSampling self, int id, str val) chgEvidence(LoopyGibbsSampling self, str nodeName, str val) chgEvidence(LoopyGibbsSampling self, int id, Vector vals) chgEvidence(LoopyGibbsSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior (*LoopyGibbsSampling self*, *int id*)
currentPosterior(LoopyGibbsSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*LoopyGibbsSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*LoopyGibbsSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*LoopyGibbsSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets(*LoopyGibbsSampling self*)
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).

eraseEvidence(*LoopyGibbsSampling self*, *int id*)
eraseEvidence(LoopyGibbsSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget(*LoopyGibbsSampling self*, *int target*)
eraseTarget(LoopyGibbsSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact(*LoopyGibbsSampling self*, *PyObject * target*, *PyObject * evs*)
Create a pyAgrum.Potential for P(targetlevs) (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(targetslevs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*LoopyGibbsSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*LoopyGibbsSampling self*, *int id*)
hasEvidence(LoopyGibbsSampling self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence(*LoopyGibbsSampling self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*LoopyGibbsSampling self*, *int id*)
hasSoftEvidence(LoopyGibbsSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

history(*LoopyGibbsSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

isDrawnAtRandom(*LoopyGibbsSampling self*)

Returns True if variables are drawn at random

Return type bool

isTarget(*LoopyGibbsSampling self*, *int variable*)
isTarget(LoopyGibbsSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

makeInference(*LoopyGibbsSampling self*)

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.

makeInference_(*LoopyGibbsSampling self*)

maxIter(*LoopyGibbsSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*LoopyGibbsSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*LoopyGibbsSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*LoopyGibbsSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*LoopyGibbsSampling self*)

Returns the number of variable drawn at each iteration

Return type int

nbrEvidence(*LoopyGibbsSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LoopyGibbsSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*LoopyGibbsSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*LoopyGibbsSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LoopyGibbsSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*LoopyGibbsSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*LoopyGibbsSampling self*, *int var*)
posterior(LoopyGibbsSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setBurnIn(*LoopyGibbsSampling self*, *int b*)

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

setDrawnAtRandom(*LoopyGibbsSampling self*, *bool _atRandom*)

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

setEpsilon(*LoopyGibbsSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*LoopyGibbsSampling self*, *int max*)

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

Raises `gum.OutOfLowerBound` – If max <= 1

setMaxTime(*LoopyGibbsSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises `gum.OutOfLowerBound` – If timeout<=0.0

setMinEpsilonRate(*LoopyGibbsSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setNbrDrawnVar(*LoopyGibbsSampling self*, *int _nbr*)

Parameters **_nbr** (*int*) – the number of variables to be drawn at each iteration

setPeriodSize(*LoopyGibbsSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*LoopyGibbsSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*LoopyGibbsSampling self*, *double vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*LoopyGibbsSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LoopyGibbsSampling self*)

Returns the list of marginal targets

Return type list

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(*LoopyGibbsSampling self*)

Returns True if the verbosity is enabled

Return type bool

Loopy Monte Carlo Sampling

class pyAgrum.**LoopyMonteCarloSampling**(*bn: pyAgrum.IBayesNet*)
Class used for inferences using a loopy version of Monte Carlo sampling.

LoopyMonteCarloSampling(bn) -> LoopyMonteCarloSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*LoopyMonteCarloSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*LoopyMonteCarloSampling self*, *int X*)
H(LoopyMonteCarloSampling self, str nodeName) -> double

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 double

addAllTargets(*LoopyMonteCarloSampling self*)
Add all the nodes as targets.

addEvidence(*LoopyMonteCarloSampling self*, *int id*, *int val*)
addEvidence(LoopyMonteCarloSampling self, str nodeName, int val) addEvidence(LoopyMonteCarloSampling self, int id, str val) addEvidence(LoopyMonteCarloSampling self, str nodeName, str val) addEvidence(LoopyMonteCarloSampling self, int id, Vector vals) addEvidence(LoopyMonteCarloSampling self, str nodeName, Vector vals)

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

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

addTarget(*LoopyMonteCarloSampling self*, *int target*)
addTarget(LoopyMonteCarloSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

chgEvidence(*LoopyMonteCarloSampling self*, *int id*, *int val*)
chgEvidence(LoopyMonteCarloSampling self, str nodeName, int val) chgEvidence(LoopyMonteCarloSampling self, int id, str val) chgEvidence(LoopyMonteCarloSampling self, str nodeName, str val) chgEvidence(LoopyMonteCarloSampling self, int id, Vector vals) chgEvidence(LoopyMonteCarloSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior(*LoopyMonteCarloSampling self*, *int id*)
currentPosterior(LoopyMonteCarloSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*LoopyMonteCarloSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*LoopyMonteCarloSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*LoopyMonteCarloSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets(*LoopyMonteCarloSampling self*)
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).

eraseEvidence(*LoopyMonteCarloSampling self*, *int id*)
eraseEvidence(LoopyMonteCarloSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises gum.IndexError – If the node does not belong to the Bayesian network

eraseTarget(*LoopyMonteCarloSampling self*, *int target*)
eraseTarget(LoopyMonteCarloSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

evidenceImpact (*LoopyMonteCarloSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes (*LoopyMonteCarloSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*LoopyMonteCarloSampling self*, *int id*)
hasEvidence(LoopyMonteCarloSampling self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence (*LoopyMonteCarloSampling self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence (*LoopyMonteCarloSampling self*, *int id*)
hasSoftEvidence(LoopyMonteCarloSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

history(*LoopyMonteCarloSampling self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

isTarget(*LoopyMonteCarloSampling self*, *int variable*)
isTarget(LoopyMonteCarloSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- `gum.IndexError` – If the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

makeInference(*LoopyMonteCarloSampling self*)
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.

makeInference_(*LoopyMonteCarloSampling self*)

maxIter(*LoopyMonteCarloSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*LoopyMonteCarloSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*LoopyMonteCarloSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*LoopyMonteCarloSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*LoopyMonteCarloSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LoopyMonteCarloSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*LoopyMonteCarloSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*LoopyMonteCarloSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LoopyMonteCarloSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*LoopyMonteCarloSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*LoopyMonteCarloSampling self*, *int var*)
posterior(LoopyMonteCarloSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon(*LoopyMonteCarloSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*LoopyMonteCarloSampling self*, *int max*)

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

Raises `gum.OutOfLowerBound` – If max <= 1

setMaxTime(*LoopyMonteCarloSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises `gum.OutOfLowerBound` – If timeout<=0.0

setMinEpsilonRate(*LoopyMonteCarloSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*LoopyMonteCarloSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*LoopyMonteCarloSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*LoopyMonteCarloSampling self*, *double vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*LoopyMonteCarloSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LoopyMonteCarloSampling self*)

Returns the list of marginal targets

Return type list

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(*LoopyMonteCarloSampling self*)

Returns True if the verbosity is enabled

Return type bool

Loopy Weighted Sampling

class pyAgrum.**LoopyWeightedSampling**(*bn: pyAgrum.IBayesNet*)
Class used for inferences using a loopy version of weighted sampling.

LoopyWeightedSampling(bn) -> LoopyWeightedSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*LoopyWeightedSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises `gum.UndefinedElement` – If no Bayes net has been assigned to the inference.

H(*LoopyWeightedSampling self*, *int X*)
H(LoopyWeightedSampling self, str nodeName) -> double

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 double

addAllTargets(*LoopyWeightedSampling self*)
Add all the nodes as targets.

addEvidence(*LoopyWeightedSampling self*, *int id*, *int val*)
addEvidence(LoopyWeightedSampling self, str nodeName, int val) addEvidence(LoopyWeightedSampling self, int id, str val) addEvidence(LoopyWeightedSampling self, str nodeName, str val) addEvidence(LoopyWeightedSampling self, int id, Vector vals) addEvidence(LoopyWeightedSampling self, str nodeName, Vector vals)

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

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

addTarget(*LoopyWeightedSampling self*, *int target*)
addTarget(LoopyWeightedSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

chgEvidence(*LoopyWeightedSampling self*, *int id*, *int val*)
chgEvidence(LoopyWeightedSampling self, str nodeName, int val) chgEvidence(LoopyWeightedSampling self, int id, str val) chgEvidence(LoopyWeightedSampling self, str nodeName, str val) chgEvidence(LoopyWeightedSampling self, int id, Vector vals) chgEvidence(LoopyWeightedSampling self, str nodeName, Vector vals)

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

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

currentPosterior(*LoopyWeightedSampling self*, *int id*)
currentPosterior(LoopyWeightedSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*LoopyWeightedSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*LoopyWeightedSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*LoopyWeightedSampling self*)
Removes all the evidence entered into the network.

eraseAllTargets(*LoopyWeightedSampling self*)
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).

eraseEvidence(*LoopyWeightedSampling self*, *int id*)
eraseEvidence(LoopyWeightedSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget(*LoopyWeightedSampling self*, *int target*)
eraseTarget(LoopyWeightedSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact(*LoopyWeightedSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes(*LoopyWeightedSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*LoopyWeightedSampling self*, *int id*)
hasEvidence(LoopyWeightedSampling self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*LoopyWeightedSampling self*, *str 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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasSoftEvidence(*LoopyWeightedSampling self*, *int id*)
hasSoftEvidence(LoopyWeightedSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

history(*LoopyWeightedSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

isTarget(*LoopyWeightedSampling self*, *int variable*)
isTarget(LoopyWeightedSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

makeInference(*LoopyWeightedSampling self*)
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.

makeInference_(*LoopyWeightedSampling self*)

maxIter(*LoopyWeightedSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*LoopyWeightedSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*LoopyWeightedSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*LoopyWeightedSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*LoopyWeightedSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LoopyWeightedSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*LoopyWeightedSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*LoopyWeightedSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LoopyWeightedSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*LoopyWeightedSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*LoopyWeightedSampling self*, *int var*)
posterior(LoopyWeightedSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon(*LoopyWeightedSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*LoopyWeightedSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*LoopyWeightedSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*LoopyWeightedSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*LoopyWeightedSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*LoopyWeightedSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*LoopyWeightedSampling self*, *double vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*LoopyWeightedSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LoopyWeightedSampling self*)

Returns the list of marginal targets

Return type list

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(*LoopyWeightedSampling self*)

Returns True if the verbosity is enabled

Return type bool

Loopy Importance Sampling

class pyAgrum.**LoopyImportanceSampling**(*bn: pyAgrum.IBayesNet*)
Class used for inferences using a loopy version of importance sampling.

LoopyImportanceSampling(bn) -> LoopyImportanceSampling

Parameters:

- **bn** (*pyAgrum.BayesNet*) – a Bayesian network

BN(*LoopyImportanceSampling self*)

Returns A constant reference over the IBayesNet referenced by this class.

Return type pyAgrum.IBayesNet

Raises gum.UndefinedElement – If no Bayes net has been assigned to the inference.

H(*LoopyImportanceSampling self*, *int X*)
H(LoopyImportanceSampling self, str nodeName) -> double

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 double

addAllTargets(*LoopyImportanceSampling self*)
Add all the nodes as targets.

addEvidence(*LoopyImportanceSampling self*, *int id*, *int val*)
addEvidence(LoopyImportanceSampling self, str nodeName, int val) addEvidence(LoopyImportanceSampling self, int id, str val) addEvidence(LoopyImportanceSampling self, str nodeName, str val) addEvidence(LoopyImportanceSampling self, int id, Vector vals) addEvidence(LoopyImportanceSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node already has an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

addTarget(*LoopyImportanceSampling self*, *int target*)
addTarget(LoopyImportanceSampling self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises gum.UndefinedElement – If target is not a NodeId in the Bayes net

chgEvidence(*LoopyImportanceSampling self*, *int id*, *int val*)

chgEvidence(LoopyImportanceSampling self, str nodeName, int val) chgEvidence(LoopyImportanceSampling self, int id, str val) chgEvidence(LoopyImportanceSampling self, str nodeName, str val) chgEvidence(LoopyImportanceSampling self, int id, Vector vals) chgEvidence(LoopyImportanceSampling self, str nodeName, Vector vals)

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

- gum.InvalidArgument – If the node does not already have an evidence
- gum.InvalidArgument – If val is not a value for the node
- gum.InvalidArgument – If the size of vals is different from the domain side of the node
- gum.FatalError – If vals is a vector of 0s
- gum.UndefinedElement – If the node does not belong to the Bayesian network

currentPosterior(*LoopyImportanceSampling self*, *int id*)

currentPosterior(LoopyImportanceSampling self, str name) -> Potential

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 ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 249) – If an element of nodes is not in targets

currentTime(*LoopyImportanceSampling self*)

Returns get the current running time in second (double)

Return type double

epsilon(*LoopyImportanceSampling self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*LoopyImportanceSampling self*)

Removes all the evidence entered into the network.

eraseAllTargets (*LoopyImportanceSampling self*)
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).

eraseEvidence (*LoopyImportanceSampling self*, *int id*)
eraseEvidence(LoopyImportanceSampling self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseTarget (*LoopyImportanceSampling self*, *int target*)
eraseTarget(LoopyImportanceSampling self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

evidenceImpact (*LoopyImportanceSampling self*, *PyObject * target*, *PyObject * 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 39)

hardEvidenceNodes (*LoopyImportanceSampling self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*LoopyImportanceSampling self*, *int id*)
hasEvidence(LoopyImportanceSampling self, str nodeName) -> bool

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 `gum.IndexError` – If the node does not belong to the Bayesian network

hasHardEvidence(*LoopyImportanceSampling self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*LoopyImportanceSampling self*, *int id*)
hasSoftEvidence(LoopyImportanceSampling self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

history(*LoopyImportanceSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

isTarget(*LoopyImportanceSampling self*, *int variable*)
isTarget(LoopyImportanceSampling self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

makeInference(*LoopyImportanceSampling self*)
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.

makeInference_(*LoopyImportanceSampling self*)

maxIter(*LoopyImportanceSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*LoopyImportanceSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*LoopyImportanceSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*LoopyImportanceSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*LoopyImportanceSampling self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*LoopyImportanceSampling self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*LoopyImportanceSampling self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*LoopyImportanceSampling self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*LoopyImportanceSampling self*)

Returns the number of marginal targets

Return type int

periodSize(*LoopyImportanceSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

posterior(*LoopyImportanceSampling self*, *int var*)
posterior(LoopyImportanceSampling self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.UndefinedElement` – If an element of nodes is not in targets

setEpsilon(*LoopyImportanceSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

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(*LoopyImportanceSampling self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*LoopyImportanceSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*LoopyImportanceSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*LoopyImportanceSampling self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

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(*LoopyImportanceSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*LoopyImportanceSampling self*, *double vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*LoopyImportanceSampling self*)

Returns the set of nodes with soft evidence

Return type set

targets(*LoopyImportanceSampling self*)

Returns the list of marginal targets

Return type list

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 (*LoopyImportanceSampling self*)

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*)

Parameters:

- **filename** (*str*) – the file to learn from

BNLearner(filename,src,parse_database=False) -> BNLearner

Parameters:

- **filename** (*str*) – the file to learn from
- **src** (*pyAgrum.BayesNet*) – the Bayesian network used to find those modalities
- **parse_database** (*bool*) – if True, the modalities specified by the user will be considered as a superset of the modalities of the variables.

BNLearner(learner) -> BNLearner

Parameters:

- **learner** (*pyAgrum.BNLearner*) – the BNLearner to copy

G2 (*BNLearner self*, *str var1*, *str var2*, *Vector_string knw={}*)

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 (*BNLearner self*, *Arc arc*)

addForbiddenArc(BNLearner self, int tail, int head) addForbiddenArc(BNLearner self, str tail, str head)

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)

addMandatoryArc(*BNLearner self*, *Arc arc*)
addMandatoryArc(BNLearner self, int tail, int head) addMandatoryArc(BNLearner self, str tail, str head)

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 gum.InvalidDirectedCycle – If the added arc creates a directed cycle in the DAG

addPossibleEdge(*BNLearner self*, *Edge edge*)
addPossibleEdge(BNLearner self, int tail, int head) addPossibleEdge(BNLearner self, str tail, str head)

chi2(*BNLearner self*, *str var1*, *str var2*, *Vector_string knw={}*)
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(*BNLearner self*)

Returns get the current running time in second (double)

Return type double

databaseWeight(*BNLearner self*)

domainSize(*BNLearner self*, *int var*)
domainSize(BNLearner self, str var) -> int

epsilon(*BNLearner self*)

Returns the value of epsilon

Return type double

eraseForbiddenArc(*BNLearner self*, *Arc arc*)
eraseForbiddenArc(BNLearner self, int tail, int head) eraseForbiddenArc(BNLearner self, str tail, str head)

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)

eraseMandatoryArc(*BNLearner self*, *Arc arc*)
eraseMandatoryArc(BNLearner self, int tail, int head) eraseMandatoryArc(BNLearner self, str tail, str head)

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)

erasePossibleEdge(*BNLearner self*, *Edge edge*)
erasePossibleEdge(BNLearner self, int tail, int head) erasePossibleEdge(BNLearner self, str tail, str head)

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)

hasMissingValues(*BNLearner self*)
Indicates whether there are missing values in the database.

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

Return type bool

history(*BNLearner self*)

Returns the scheme history

Return type tuple

Raises `gum.OperationNotAllowed` – If the scheme did not performed or if verbosity is set to false

idFromName(*BNLearner self*, *str var_name*)

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

Returns the column id corresponding to a variable name

Return type int

Raises `gum.MissingVariableInDatabase` – If a variable of the BN is not found in the database.

latentVariables(*BNLearner self*)
latentVariables(BNLearner self) -> vector< pyAgrum.Arc,allocator< pyAgrum.Arc > > const

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

Returns the list of latent variables

Return type list

learnBN (*BNLearner self*)
learn a BayesNet from a file (must have read the db before)

Returns the learned BayesNet

Return type *pyAgrum.BayesNet* (page 48)

learnDAG (*BNLearner self*)
learn a structure from a file

Returns the learned DAG

Return type *pyAgrum.DAG* (page 7)

learnMixedStructure (*BNLearner self*)

Warning: learner must be using 3off2 or MIIC algorithm

Returns the learned structure as an EssentialGraph

Return type *pyAgrum.EssentialGraph* (page 66)

learnParameters (*BNLearner self*, *DAG dag*, *bool take_into_account_score=True*)
learnParameters(BNLearner self, bool take_into_account_score=True) -> BayesNet

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

Parameters

- **dag** (pyAgrum.DAG (page 7)) –
- **bn** (pyAgrum.BayesNet (page 48)) –
- **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 48)

Raises

- gum.MissingVariableInDatabase – If a variable of the BN is not found in the database
- gum.UnknownLabelInDatabase – If a label is found in the database that do not correspond to the variable

logLikelihood (*BNLearner self*, *vector< int, allocator< int > > vars*, *vector< int, allocator< int > > knowing={}*)
logLikelihood(BNLearner self, vector< int,allocator< int > > vars) -> double logLikelihood(BNLearner self, Vector_string vars, Vector_string knowing={}) -> double logLikelihood(BNLearner self, Vector_string vars) -> double

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 double

maxIter(*BNLearner self*)

Returns the criterion on number of iterations

Return type int

maxTime(*BNLearner self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*BNLearner self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*BNLearner self*)

Returns the value of the minimal epsilon rate

Return type double

nameFromId(*BNLearner self*, *int id*)

Parameters **id** – a node id

Returns the variable's name

Return type str

names(*BNLearner self*)

Returns the names of the variables in the database

Return type List[str]

nbCols(*BNLearner self*)
Return the nimber of columns in the database

Returns the number of columns in the database

Return type int

nbRows(*BNLearner self*)
Return the number of row in the database

Returns the number of rows in the database

Return type int

nbrIterations(*BNLearner self*)

Returns the number of iterations

Return type int

periodSize(*BNLearner self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – 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(*BNLearner self*, *vector< int, allocator< int > > vars*)
rawPseudoCount(BNLearner self, Vector_string vars) -> Vector

recordWeight(*BNLearner self*, *size_t i*)

setAprioriWeight(*weight*)
Deprecated methods in BNLearner for pyAgrum>0.14.0

setDatabaseWeight(*BNLearner self*, *double new_weight*)
Set the database weight.

Parameters **weight** (*double*) – the database weight

setEpsilon(*BNLearner self*, *double eps*)

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

Raises gum.OutOfLowerBound – If eps<0

setInitialDAG(*BNLearner self*, *DAG g*)

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

setMaxIndegree(*BNLearner self*, *int max_indegree*)

setMaxIter(*BNLearner self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*BNLearner self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*BNLearner self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*BNLearner self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

setPossibleSkeleton(*BNLearner self*, *UndiGraph skeleton*)

setRecordWeight(*BNLearner self*, *size_t i*, *double weight*)

setSliceOrder(*BNLearner self*, *PyObject * l*)
setSliceOrder(BNLearner self, pyAgrum.NodeProperty< int > slice_order) setSliceOrder(BNLearner self, vector< vector< str,allocator< str > >,allocator< vector< str,allocator< str > > > > slices)

Set a partial order on the nodes.

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

setVerbosity(*BNLearner self*, *bool v*)

Parameters **v** (*bool*) – verbosity

use3off2(*BNLearner self*)
Indicate that we wish to use 3off2.

useAprioriBDeu(*BNLearner self*, *double weight=1*)
useAprioriBDeu(BNLearner self)

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** (*double*) – the apriori weight

useAprioriDirichlet(*BNLearner self*, *str filename*, *double weight=1*)
useAprioriDirichlet(BNLearner self, str filename)

useAprioriSmoothing(*BNLearner self*, *double weight=1*)
useAprioriSmoothing(BNLearner self)

useEM(*BNLearner self*, *double epsilon*)
Indicates if we use EM for parameter learning.

Parameters **epsilon** (*double*) – 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.

useGreedyHillClimbing(*BNLearner self*)

useK2(*BNLearner self*, *PyObject * l*)
useK2(BNLearner self, pyAgrum.Sequence< int > order) useK2(BNLearner self, vector< int,allocator< int > > order)

Indicate that we wish to use K2.

Parameters **order** (*list*) – a list of ids

useLocalSearchWithTabuList(*BNLearner self*, *int tabu_size=100*, *int nb_decrease=2*)
useLocalSearchWithTabuList(BNLearner self, int tabu_size=100) useLocalSearchWithTabuList(BNLearner self)

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

useMDL(*BNLearner self*)
Indicate that we wish to use the MDL correction for 3off2 or MIIC

useMIIC(*BNLearner self*)
Indicate that we wish to use MIIC.

useNML(*BNLearner self*)
Indicate that we wish to use the NML correction for 3off2 or MIIC

useNoApriori(*BNLearner self*)

useNoCorr(*BNLearner self*)
Indicate that we wish to use the NoCorr correction for 3off2 or MIIC

useScoreAIC(*BNLearner self*)

useScoreBD(*BNLearner self*)

useScoreBDeu(*BNLearner self*)

useScoreBIC(*BNLearner self*)

useScoreK2(*BNLearner self*)

useScoreLog2Likelihood(*BNLearner self*)

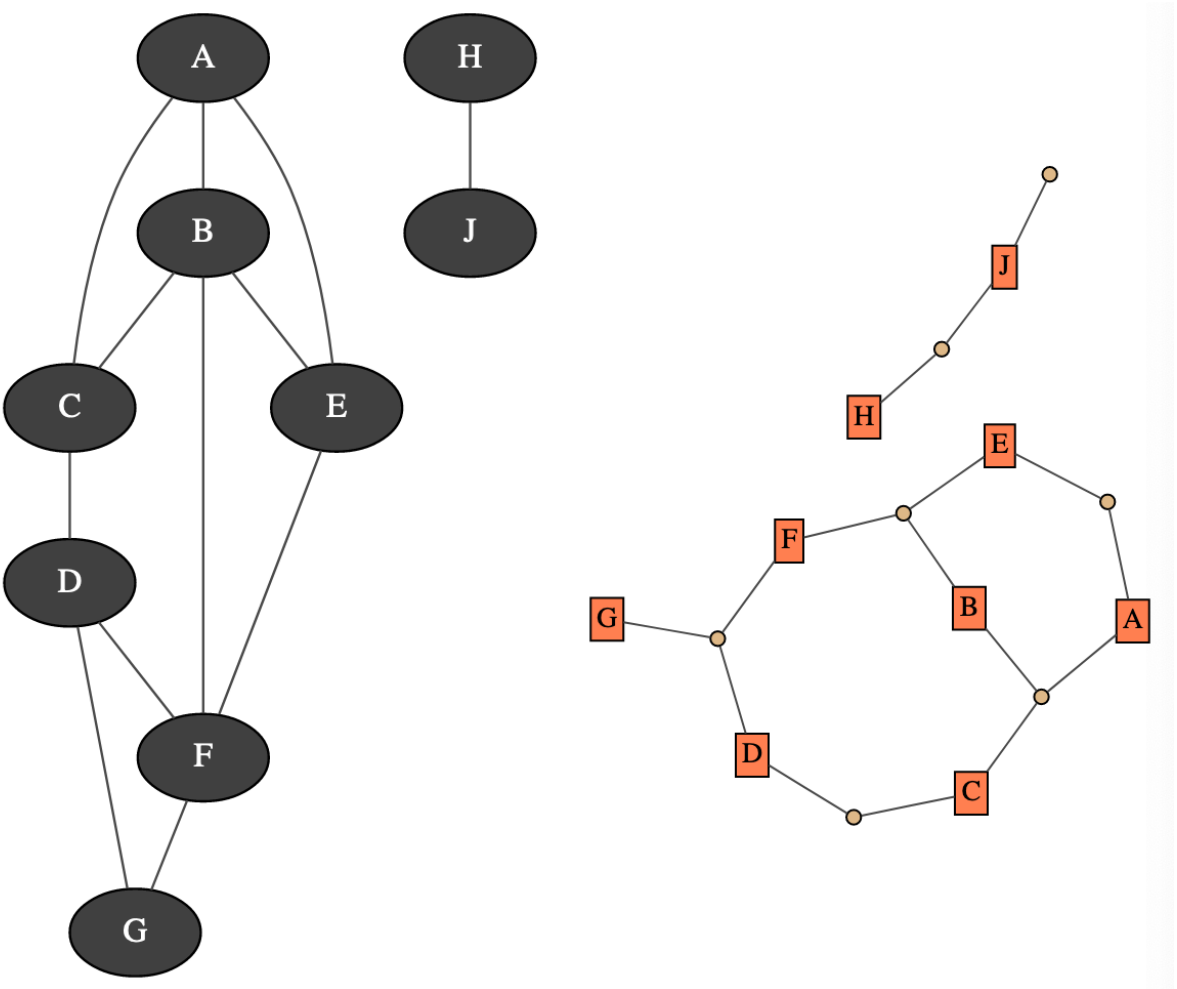
verbosity(*BNLearner self*)

Returns True if the verbosity is enabled

Return type bool

CHAPTER 5

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/35-MarkovNetwork.ipynb.html)

Reference

5.1 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(*MarkovNet self*, *DiscreteVariable var*)

add(MarkovNet self, str name, unsigned int nbrmod) -> int add(MarkovNet self, DiscreteVariable var, int id) -> int

Add a variable to the pyAgrum.MarkovNet.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – 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

- gum.DuplicateLabel – If variable.name() is already used in this pyAgrum.MarkovNet.
- gum.NotAllowed – If nbrmod is less than 2
- gum.DuplicateElement – If id is already used.

addFactor(*MarkovNet self*, *Vector_string varnames*)

addFactor(MarkovNet self, Set vars) -> Potential addFactor(MarkovNet self, Potential factor) -> Potential addFactor(MarkovNet self, PyObject * seq) -> Potential

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 (*MarkovNet self*)

changeVariableLabel (*MarkovNet self*, *int id*, *str old_label*, *str new_label*)
changeVariableLabel(MarkovNet self, str name, str old_label, str new_label)

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 gum.NotFound – if id/name is not a variable or if old_label does not exist.

changeVariableName (*MarkovNet self*, *int id*, *str new_name*)
changeVariableName(MarkovNet self, str name, str new_name)

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

- gum.DuplicateLabel – If new_name is already used in this MarkovNet.
- gum.NotFound – If no variable matches id.

clear (*MarkovNet self*)
Clear the whole MarkovNet

completeInstantiation (*GraphicalModel self*)

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 (*IMarkovNet self*)

edges (*MarkovNet self*)

empty(*GraphicalModel self*)

endTopologyTransformation(*MarkovNet self*)
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 158)

erase(*MarkovNet self*, *int varId*)
erase(MarkovNet self, str name) erase(MarkovNet self, DiscreteVariable var)

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 21)) – A reference on the variable to remove.

eraseFactor(*MarkovNet self*, *Set vars*)
eraseFactor(MarkovNet self, Vector_string varnames) eraseFactor(MarkovNet self, PyObject * seq)

exists(*UGmodel self*, *int node*)

existsEdge(*UGmodel self*, *int node1*, *int node2*)
existsEdge(UGmodel self, str name1, str name2) -> bool

factor(*MarkovNet self*, *Set varIds*)
factor(MarkovNet self, Vector_string varnames) -> Potential factor(MarkovNet self, PyObject * nodeseq) -> Potential

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 39)

Raises gum.NotFound – If no variable's id matches varId.

factors(*MarkovNet self*)

static fastPrototype(*str dotlike*, *int 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=gum.MarkovNet.fastPrototype('A-B[1,3]-C{yes|No}-D[2,4]-E[1,2.5,3.9]
↪',6)
```

(continues on next page)

(continued from previous page)

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 158)

static fromBN(*BayesNet bn*)

generateFactor(*MarkovNet self*, *Set 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.

generateFactors(*MarkovNet self*)
Randomly generates factors parameters for a given structure.

graph(*MarkovNet self*)

hasSameStructure(*UGmodel self*, *UGmodel other*)

idFromName(*MarkovNet self*, *str name*)

ids(*GraphicalModel self*, *Vector_string names*)

isIndependent(*MarkovNet self*, *PyObject * X*, *PyObject * Y*, *PyObject * Z*)

loadUAI(*MarkovNet self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load an UAI file.

Parameters

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

log10DomainSize(*MarkovNet self*)

maxNonOneParam(*IMarkovNet self*)

maxParam(*IMarkovNet self*)

maxVarDomainSize(*IMarkovNet self*)

minNonZeroParam(*IMarkovNet self*)

minParam(*IMarkovNet self*)

minimalCondSet(*MarkovNet self*, *int target*, *PyObject * list*)
minimalCondSet(MarkovNet self, PyObject * targets, PyObject * list) -> PyObject *

names(*MarkovNet self*)

neighbours(*MarkovNet self*, *PyObject * norid*)

nodeId(*MarkovNet self*, *DiscreteVariable var*)

nodes(*MarkovNet self*)

nodeset (*GraphicalModel self*, *Vector_string names*)

property (*GraphicalModel self*, *str name*)

propertyWithDefault (*GraphicalModel self*, *str name*, *str byDefault*)

saveUAI (*MarkovNet self*, *str name*)
Save the MarkovNet in an UAI file.

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

setProperty (*GraphicalModel self*, *str name*, *str value*)

size (*MarkovNet self*)

sizeEdges (*UGmodel self*)

smallestFactorFromNode (*MarkovNet self*, *int node*)

toDot (*IMarkovNet self*)

toDotAsFactorGraph (*IMarkovNet self*)

variable (*MarkovNet self*, *int id*)
variable(MarkovNet self, str name) -> DiscreteVariable

variableFromName (*MarkovNet self*, *str name*)

variableNodeMap (*MarkovNet self*)

5.2 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.

5.2.1 Shafer Shenoy Inference

class pyAgrum.**ShaferShenoyMNInference**(*MN: pyAgrum.IMarkovNet*, *use_binary_join_tree: bool = True*)
Class used for Shafer-Shenoy inferences for Markov network.

ShaferShenoyInference(bn) -> ShaferShenoyInference

Parameters:

- **mn** (*pyAgrum.MarkovNet*) – a Markov network

H (*ShaferShenoyMNInference self*, *int X*)
H(ShaferShenoyMNInference self, str nodeName) -> double

Parameters

- **X** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns the Shanon's entropy of a node given the observation

Return type double

I (*ShaferShenoyMNInference self*, *int X*, *int 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 double

MN(*ShaferShenoyMNInference self*)

VI(*ShaferShenoyMNInference self*, *int X*, *int 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 double

addAllTargets(*ShaferShenoyMNInference self*)
Add all the nodes as targets.

addEvidence(*ShaferShenoyMNInference self*, *int id*, *int val*)
addEvidence(ShaferShenoyMNInference self, str nodeName, int val) addEvidence(ShaferShenoyMNInference self, int id, str val) addEvidence(ShaferShenoyMNInference self, str nodeName, str val) addEvidence(ShaferShenoyMNInference self, int id, Vector vals) addEvidence(ShaferShenoyMNInference self, str nodeName, Vector vals)

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

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

addJointTarget(*ShaferShenoyMNInference self*, *PyObject * 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

Raises `gum.UndefinedElement` – If some node(s) do not belong to the Bayesian network

addTarget(*ShaferShenoyMNInference self*, *int target*)
addTarget(ShaferShenoyMNInference self, str nodeName)

Add a marginal target to the list of targets.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Raises `gum.UndefinedElement` – If target is not a NodeId in the Bayes net

chgEvidence(*ShaferShenoyMNInference self*, *int id*, *int val*)
chgEvidence(ShaferShenoyMNInference self, str nodeName, int val) chgEvidence(ShaferShenoyMNInference self, int id, str val) chgEvidence(ShaferShenoyMNInference self, str nodeName, str val) chgEvidence(ShaferShenoyMNInference self, int id, Vector vals) chgEvidence(ShaferShenoyMNInference self, str nodeName, Vector vals)

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

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

eraseAllEvidence(*ShaferShenoyMNInference self*)
Removes all the evidence entered into the network.

eraseAllJointTargets(*ShaferShenoyMNInference self*)
Clear all previously defined joint targets.

eraseAllMarginalTargets(*ShaferShenoyMNInference self*)
Clear all the previously defined marginal targets.

eraseAllTargets(*ShaferShenoyMNInference self*)
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).

eraseEvidence(*ShaferShenoyMNInference self*, *int id*)
eraseEvidence(ShaferShenoyMNInference self, str nodeName)

Remove the evidence, if any, corresponding to the node Id or name.

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises `gum.IndexError` – If the node does not belong to the Bayesian network

eraseJointTarget(*ShaferShenoyMNInference self*, *PyObject * targets*)
Remove, if existing, the joint target.

Parameters **list** – a list of names or Ids of nodes

Raises

- `gum.IndexError` – If one of the node does not belong to the Bayesian network
- `gum.UndefinedElement` – If node Id is not in the Bayesian network

eraseTarget(*ShaferShenoyMNInference self*, *int target*)
eraseTarget(ShaferShenoyMNInference self, str nodeName)

Remove, if existing, the marginal target.

Parameters

- **target** (*int*) – a node Id
- **nodeName** (*int*) – a node name

Raises

- gum.IndexError – If one of the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

evidenceImpact(*ShaferShenoyMNInference self*, *PyObject * target*, *PyObject * 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 39)

evidenceJointImpact(*ShaferShenoyMNInference self*, *PyObject * targets*, *PyObject * evs*)
evidenceJointImpact(ShaferShenoyMNInference self, Vector_string targets, Vector_string evs) -> Potential

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 39)

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

evidenceProbability(*ShaferShenoyMNInference self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes(*ShaferShenoyMNInference self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*ShaferShenoyMNInference self*, *int id*)
hasEvidence(ShaferShenoyMNInference self, str nodeName) -> bool

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 gum.IndexError – If the node does not belong to the Bayesian network

hasHardEvidence(*ShaferShenoyMNInference self*, *str 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 gum.IndexError – If the node does not belong to the Bayesian network

hasSoftEvidence(*ShaferShenoyMNInference self*, *int id*)
hasSoftEvidence(ShaferShenoyMNInference self, str nodeName) -> bool

Parameters

- **id** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if node has received a soft evidence

Return type bool

Raises gum.IndexError – If the node does not belong to the Bayesian network

isJointTarget(*ShaferShenoyMNInference self*, *PyObject * targets*)

Parameters **list** – a list of nodes ids or names.

Returns True if target is a joint target.

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

isTarget(*ShaferShenoyMNInference self*, *int variable*)
isTarget(ShaferShenoyMNInference self, str nodeName) -> bool

Parameters

- **variable** (*int*) – a node Id
- **nodeName** (*str*) – a node name

Returns True if variable is a (marginal) target

Return type bool

Raises

- gum.IndexError – If the node does not belong to the Bayesian network
- gum.UndefinedElement – If node Id is not in the Bayesian network

joinTree(*ShaferShenoyMNInference self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*ShaferShenoyMNInference self*, *PyObject * targets*)

jointPosterior(*ShaferShenoyMNInference self*, *PyObject * 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 ref to the posterior joint probability of the set of nodes.

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – If an element of nodes is not in targets

jointTargets(*ShaferShenoyMNInference self*)

Returns the list of target sets

Return type list

junctionTree(*ShaferShenoyMNInference self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*ShaferShenoyMNInference self*)
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.

nbrEvidence(*ShaferShenoyMNInference self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*ShaferShenoyMNInference self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*ShaferShenoyMNInference self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*ShaferShenoyMNInference self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*ShaferShenoyMNInference self*)

Returns the number of marginal targets

Return type int

posterior(*ShaferShenoyMNInference self*, *int var*)
posterior(ShaferShenoyMNInference self, str nodeName) -> Potential posterior(ShaferShenoyMNInference self, str nodeName) -> Potential

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 ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises gum.UndefinedElement – 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

setTriangulation(*ShaferShenoyMNInference self*, *Triangulation new_triangulation*)

softEvidenceNodes(*ShaferShenoyMNInference self*)

Returns the set of nodes with soft evidence

Return type set

targets(*ShaferShenoyMNInference self*)

Returns the list of marginal targets

Return type list

updateEvidence(*evidces*)

Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

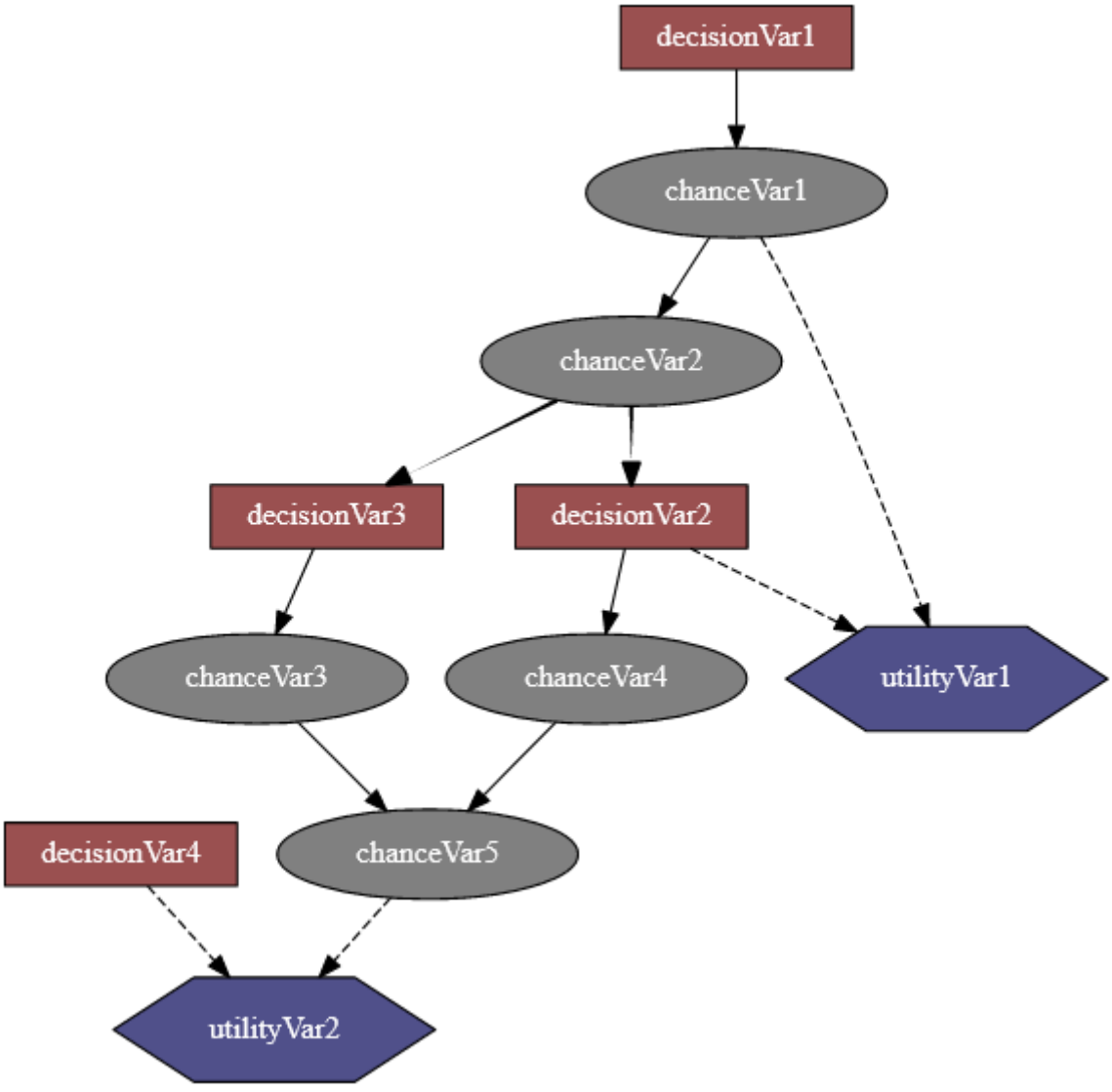
Parameters **evidces** (*dict*) – a dict of evidences

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 6

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/34-InfluenceDiagram.ipynb.html)

Reference

6.1 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(*InfluenceDiagram self*, *DiscreteVariable variable*, *int 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 21)) – 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 gum.DuplicateElement – If id(<>0) is already used

addArc(*InfluenceDiagram self*, *int tail*, *int head*)

addArc(InfluenceDiagram self, str tail, str head)

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

- gum.InvalidEdge – If arc.tail and/or arc.head are not in the ID.
- gum.InvalidEdge – If tail is a utility node

addChanceNode(*InfluenceDiagram self*, *DiscreteVariable variable*, *int id=0*)
addChanceNode(InfluenceDiagram self, DiscreteVariable variable, pyAgrum.MultiDimImplementation aContent, int id=0) -> int

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 21)) – 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 gum.DuplicateElement – If id(<>0) is already used

addDecisionNode(*InfluenceDiagram self*, *DiscreteVariable variable*, *int id=0*)
Add a decision variable.

The id of the new variable is automatically generated.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – 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 gum.DuplicateElement – If id(<>0) is already used

addUtilityNode(*InfluenceDiagram self*, *DiscreteVariable variable*, *int id=0*)
addUtilityNode(InfluenceDiagram self, DiscreteVariable variable, pyAgrum.MultiDimImplementation aContent, int id=0) -> int

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 21)) – 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

- `gum.InvalidArgument` – If variable has more than one label
- `gum.DuplicateElement` – If id(<>0) is already used

ancestors (*DAGmodel self*, *int id*)
ancestors(DAGmodel self, str name) -> Set

arcs (*InfluenceDiagram self*)

Returns the list of all the arcs in the Influence Diagram.

Return type list

chanceNodeSize (*InfluenceDiagram self*)

Returns the number of chance nodes.

Return type int

changeVariableName (*InfluenceDiagram self*, *int id*, *str new_name*)
changeVariableName(InfluenceDiagram self, str name, str new_name)

Parameters

- **id** (*int*) – the node Id
- **new_name** (*str*) – the name of the variable

Raises

- `gum.DuplicateLabel` – If this name already exists
- `gum.NotFound` – If no nodes matches id.

children (*InfluenceDiagram self*, *int id*)

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

Returns the set of all the children

Return type Set

clear (*InfluenceDiagram self*)

completeInstantiation (*GraphicalModel self*)

cpt (*InfluenceDiagram self*, *int varId*)
cpt(InfluenceDiagram self, str name) -> Potential

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 39)

Raises `gum.NotFound` – If no variable's id matches varId.

dag (*DAGmodel self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

decisionNodeSize (*InfluenceDiagram self*)

Returns the number of decision nodes

Return type int

decisionOrder (*InfluenceDiagram self*)

decisionOrderExists (*InfluenceDiagram self*)

Returns True if a directed path exist with all decision node

Return type bool

descendants(*DAGmodel self*, *int id*)
descendants(DAGmodel self, str name) -> Set

empty(*GraphicalModel self*)

erase(*InfluenceDiagram self*, *int id*)
erase(InfluenceDiagram self, str name) erase(InfluenceDiagram self, DiscreteVariable var)

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 21)) – The reference on the variable to remove.

eraseArc(*InfluenceDiagram self*, *Arc arc*)
eraseArc(InfluenceDiagram self, int tail, int head) eraseArc(InfluenceDiagram self, str tail, str head)

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

exists(*DAGmodel self*, *int node*)

existsArc(*DAGmodel self*, *int tail*, *int head*)
existsArc(DAGmodel self, str nametail, str namehead) -> bool

existsPathBetween(*InfluenceDiagram self*, *int src*, *int dest*)
existsPathBetween(InfluenceDiagram self, str src, str dest) -> bool

Returns true if a path exists between two nodes.

Return type bool

family(*DAGmodel self*, *int id*)
family(DAGmodel self, str name) -> Set

static fastPrototype(*str dotlike*, *int 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 gum.RangeVariable using the default domain size (second argument)
  - with *'a[10]'*, the variable is a gum.RangeVariable using 10 as domain size (from 0 to 9)
  - with *'a[3,7]'*, the variable is a gum.RangeVariable using a domainSize from 3 to 7

- with *'a[1,3.14,5,6.2]'*, the variable is a gum.DiscretizedVariable using the given ticks (at least 3 values)
- with *'a{top|middle|bottom}'*, the variable is a gum.LabelizedVariable using the given labels.

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=gum.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 170)

getDecisionGraph (*InfluenceDiagram self*)

Returns the temporal Graph.

Return type *pyAgrum.DAG* (page 7)

hasSameStructure (*DAGmodel self*, *DAGmodel other*)

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

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

Return type bool

idFromName (*InfluenceDiagram self*, *str 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 gum.NotFound – If no such name exists in the graph.

ids (*GraphicalModel self*, *Vector_string names*)

isChanceNode (*InfluenceDiagram self*, *int varId*)
isChanceNode(InfluenceDiagram self, str name) -> bool

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

Returns true if node is a chance node

Return type bool

isDecisionNode (*InfluenceDiagram self*, *int varId*)
isDecisionNode(InfluenceDiagram self, str name) -> bool

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

Returns true if node is a decision node

Return type bool

isIndependent (*DAGmodel self*, *int X*, *int Y*, *Set Z*)
isIndependent(DAGmodel self, str Xname, str Yname, Vector_string Zanmes) -> bool isIndependent(DAGmodel self, Set X, Set Y, Set Z) -> bool isIndependent(DAGmodel self, Vector_string Xname, Vector_string Yname, Vector_string Zanmes) -> bool

isUtilityNode (*InfluenceDiagram self*, *int varId*)
isUtilityNode(InfluenceDiagram self, str name) -> bool

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

Returns true if node is an utility node

Return type bool

loadBIFXML (*InfluenceDiagram self*, *str name*, *PyObject * l=(PyObject *) 0*)
Load a BIFXML file.

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

Raises

- gum.IOError – If file not found
- gum.FatalError – If file is not valid

log10DomainSize (*GraphicalModel self*)

moralGraph (*DAGmodel self*, *bool 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 9)

moralizedAncestralGraph (*DAGmodel self*, *Set nodes*)
moralizedAncestralGraph(DAGmodel self, Vector_string nodenames) -> UndiGraph

names (*InfluenceDiagram self*)

Returns The names of the InfluenceDiagram variables

Return type list

nodeId (*InfluenceDiagram self*, *DiscreteVariable var*)

Parameters **var** (pyAgrum.DiscreteVariable (page 21)) – a variable

Returns the id of the variable

Return type int

Raises gum.IndexError – If the InfluenceDiagram does not contain the variable

nodes (*InfluenceDiagram self*)

Returns the set of ids

Return type set

nodeset (*GraphicalModel self*, *Vector_string names*)

parents (*InfluenceDiagram self*, *int id*)

Parameters **id** – The id of the child node

Returns the set of the parents ids.

Return type set

property (*GraphicalModel self*, *str name*)

propertyWithDefault (*GraphicalModel self*, *str name*, *str byDefault*)

saveBIFXML (*InfluenceDiagram self*, *str name*)
Save the BayesNet in a BIFXML file.

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

setProperty (*GraphicalModel self*, *str name*, *str value*)

size (*DAGmodel self*)

Returns the number of nodes in the graph

Return type int

sizeArcs (*DAGmodel self*)

Returns the number of arcs in the graph

Return type int

toDot (*InfluenceDiagram self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder (*DAGmodel self*, *bool clear=True*)

Returns the list of the nodes Ids in a topological order

Return type List

Raises `gum.InvalidDirectedCycle` – If this graph contains cycles

utility (*InfluenceDiagram self*, *int varId*)
utility(InfluenceDiagram self, str name) -> Potential

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

Returns the utility table of the node

Return type *pyAgrum.Potential* (page 39)

Raises `gum.IndexError` – If the InfluenceDiagram does not contain the variable

utilityNodeSize (*InfluenceDiagram self*)

Returns the number of utility nodes

Return type int

variable (*InfluenceDiagram self*, *int id*)

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

Returns a constant reference over a variabe given it's node id

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises `gum.NotFound` – If no variable's id matches the parameter

variableFromName (*InfluenceDiagram self*, *str name*)

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

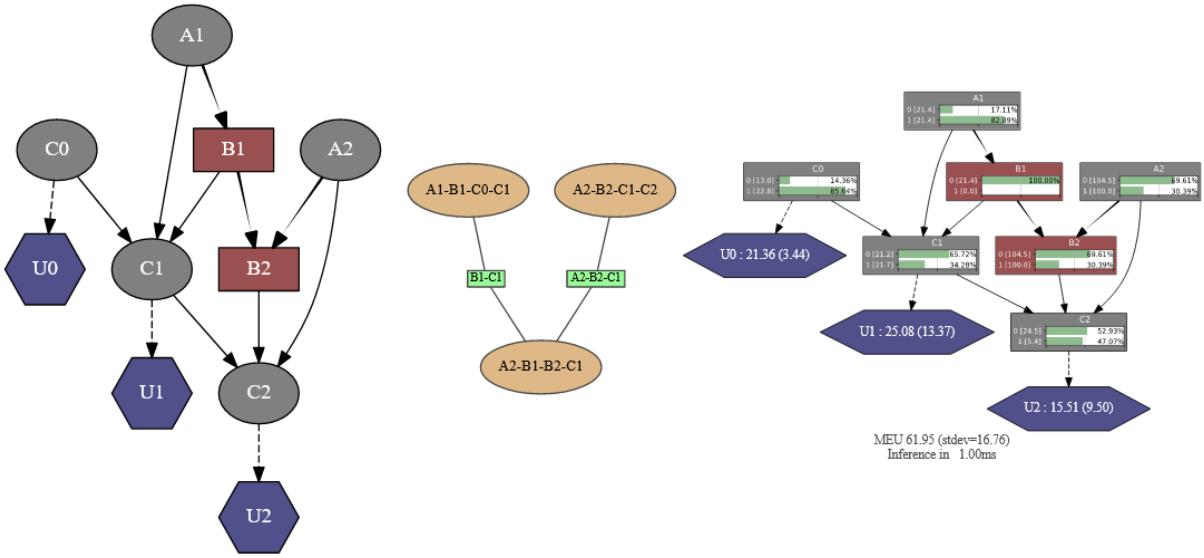
Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises `gum.IndexError` – If the InfluenceDiagram does not contain the variable

variableNodeMap (*GraphicalModel self*)

6.2 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: pyAgrum.InfluenceDiagram*)

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).

MEU(*ShaferShenoyLIMIDInference self*)

MEU(ShaferShenoyLIMIDInference self) -> PyObject *

Returns maximum expected utility obtained from inference.

Raises gum.OperationNotAllowed – If no inference have yet been made

addEvidence(*ShaferShenoyLIMIDInference self*, *int id*, *int val*)

addEvidence(ShaferShenoyLIMIDInference self, str nodeName, int val) addEvidence(ShaferShenoyLIMIDInference self, int id, str val) addEvidence(ShaferShenoyLIMIDInference self, str nodeName, str val) addEvidence(ShaferShenoyLIMIDInference self, int id, Vector vals) addEvidence(ShaferShenoyLIMIDInference self, str nodeName, Vector vals)

addNoForgettingAssumption(*ShaferShenoyLIMIDInference self*, *vector< int, allocator< int > > ids*)

addNoForgettingAssumption(ShaferShenoyLIMIDInference self, Vector_string names)

chgEvidence(*ShaferShenoyLIMIDInference self*, *int id*, *int val*)

chgEvidence(ShaferShenoyLIMIDInference self, str nodeName, int val) chgEvidence(ShaferShenoyLIMIDInference self, int id, str val) chgEvidence(ShaferShenoyLIMIDInference self, str nodeName, str val) chgEvidence(ShaferShenoyLIMIDInference self, int id, Vector vals) chgEvidence(ShaferShenoyLIMIDInference self, str nodeName, Vector vals)

clear(*ShaferShenoyLIMIDInference self*)

eraseAllEvidence(*ShaferShenoyLIMIDInference self*)

Removes all the evidence entered into the diagram.

eraseEvidence(*ShaferShenoyLIMIDInference self*, *int id*)
eraseEvidence(ShaferShenoyLIMIDInference self, str nodeName)

Parameters **evidence** (pyAgrum.Potential (page 39)) – the evidence to remove

Raises gum.IndexError – If the evidence does not belong to the influence diagram

hardEvidenceNodes(*ShaferShenoyLIMIDInference self*)

hasEvidence(*ShaferShenoyLIMIDInference self*, *int id*)
hasEvidence(ShaferShenoyLIMIDInference self, str nodeName) -> bool

hasHardEvidence(*ShaferShenoyLIMIDInference self*, *str nodeName*)

hasNoForgettingAssumption(*ShaferShenoyLIMIDInference self*)

hasSoftEvidence(*ShaferShenoyLIMIDInference self*, *int id*)
hasSoftEvidence(ShaferShenoyLIMIDInference self, str nodeName) -> bool

influenceDiagram(*ShaferShenoyLIMIDInference self*)
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 170)

isSolvable(*ShaferShenoyLIMIDInference self*)

junctionTree(*ShaferShenoyLIMIDInference self*)

makeInference(*ShaferShenoyLIMIDInference self*)
Makes the inference.

meanVar(*ShaferShenoyLIMIDInference self*, *int node*)
meanVar(ShaferShenoyLIMIDInference self, str name) -> pair< double,double > meanVar(ShaferShenoyLIMIDInference self, int node) -> PyObject *

nbrEvidence(*ShaferShenoyLIMIDInference self*)

nbrHardEvidence(*ShaferShenoyLIMIDInference self*)

nbrSoftEvidence(*ShaferShenoyLIMIDInference self*)

optimalDecision(*ShaferShenoyLIMIDInference self*, *int decisionId*)
optimalDecision(ShaferShenoyLIMIDInference self, str decisionName) -> Potential

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

- gum.OperationNotAllowed – If no inference have yet been made
- gum.InvalidNode – If node given in parmaeter is not a decision node

posterior(*ShaferShenoyLIMIDInference self*, *int node*)
posterior(ShaferShenoyLIMIDInference self, str name) -> Potential posterior(ShaferShenoyLIMIDInference self, int var) -> Potential posterior(ShaferShenoyLIMIDInference self, str nodeName) -> Potential

posteriorUtility(*ShaferShenoyLIMIDInference self*, *int node*)
posteriorUtility(ShaferShenoyLIMIDInference self, str name) -> Potential

reducedGraph(*ShaferShenoyLIMIDInference self*)

reducedLIMID(*ShaferShenoyLIMIDInference self*)

reversePartialOrder(*ShaferShenoyLIMIDInference self*)

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 (*ShaferShenoyLIMIDInference self*)

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 7

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

aggType

a(9).str

min/max/count/exists/forall/or/and/amplitude/median

Type aggType

classAggregates(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of aggregates in the class

Return type list

Raises gum.IndexError – If the class is not in the PRM

classAttributes(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of attributes

Return type list

Raises gum.IndexError – If the class is not in the PRM

classDag(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns a description of the DAG

Return type tuple

Raises gum.IndexError – If the class is not in the PRM

classImplements(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of interfaces implemented by the class

Return type list

classParameters(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of parameters

Return type list

Raises gum.IndexError – If the class is not in the PRM

classReferences(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of references

Return type list

Raises gum.IndexError – If the class is not in the PRM

classSlotChains(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of class slot chains

Return type list

Raises gum.IndexError – if the class is not in the PRM

classes(*PRMexplorer self*)

Returns the list of classes

Return type list

cpf(*PRMexplorer self*, *str class_name*, *str attribute*)

Parameters

- **class_name** (*str*) – a class name
- **attribute** (*str*) – an attribute

Returns the potential of the attribute

Return type *pyAgrum.Potential* (page 39)

Raises

- gum.OperationNotAllowed – If the class element doesn't have any pyAgrum.Potential (like a pyAgrum.PRMReferenceSlot).
- gum.IndexError – If the class is not in the PRM
- gum.IndexError – If the attribute in parameters does not exist

getDirectSubClass(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of direct subclasses

Return type list

Raises gum.IndexError – If the class is not in the PRM

getDirectSubInterfaces(*PRMexplorer self*, *str interface_name*)

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

Returns the list of direct subinterfaces

Return type list

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

getDirectSubTypes(*PRMexplorer self*, *str type_name*)

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

Returns the list of direct subtypes

Return type list

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

getImplementations(*PRMexplorer self*, *str interface_name*)

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

Returns the list of classes implementing the interface

Return type str

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

getLabelMap(*PRMexplorer self*, *str type_name*)

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

Returns a dict containing pairs of label and their values

Return type dict

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

getLabels(*PRMexplorer self*, *str type_name*)

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

Returns the list of type labels

Return type list

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

getSuperClass(*PRMexplorer self*, *str class_name*)

Parameters **class_name** (*str*) – a class name

Returns the class extended by class_name

Return type str

Raises `gum.IndexError` – If the class is not in the PRM

getSuperInterface(*PRMexplorer self*, *str interface_name*)

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

Returns the interace extended by interface_name

Return type str

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

getSuperType(*PRMexplorer self*, *str type_name*)

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

Returns the type extended by type_name

Return type str

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

getalltheSystems(*PRMexplorer self*)

Returns the list of all the systems and their components

Return type list

interAttributes(*PRMexplorer self*, *str interface_name*, *bool 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 gum.IndexError – If the type is not in the PRM

interReferences(*PRMexplorer self*, *str 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 gum.IndexError – If the type is not in the PRM

interfaces(*PRMexplorer self*)

Returns the list of interfaces in the PRM

Return type list

isAttribute(*PRMexplorer self*, *str class_name*, *str 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

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

isClass(*PRMexplorer self*, *str name*)

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

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

Return type bool

isInterface(*PRMexplorer self*, *str name*)

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

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

Return type bool

isType(*PRMexplorer self*, *str name*)

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

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

Return type bool

load(*PRMexplorer self*, *str filename*, *str classpath=""*, *bool verbose=False*)
Load a PRM into the explorer.

Parameters

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

Raises gum.FatalError – If file not found

types(*PRMexplorer self*)

Returns the list of the custom types in the PRM

Return type list

CHAPTER 8

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/33-credalNetworks.ipynb.html)

Reference

8.1 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 48)) – The path to a BayesNet or the BN itself which contains lower probabilities.
- **src_max_den** (*str or* pyAgrum.BayesNet (page 48)) – The (optional) path to a BayesNet or the BN itself which contains upper probabilities.

addArc(*CredalNet self*, *int tail*, *int head*)

Adds an arc between two nodes

Parameters

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

Raises

- gum.InvalidDirectedCircle – If any (directed) cycle is created by this arc

- `gum.InvalidNode` – If head or tail does not belong to the graph nodes
- `gum.DuplicateElement` – If one of the arc already exists

addVariable(*CredalNet self*, *str name*, *int 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(*CredalNet self*)

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

bnToCredal(*CredalNet self*, *double beta*, *bool oneNet*, *bool 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** (*double*) – 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

computeCPTMinMax(*CredalNet self*)

Used with binary networks to speed-up L2U inference.

Store the lower and upper probabilities of each node X over the 'True' modality.

credalNet_currentCpt(*CredalNet self*)

Warning: Experimental function - Return type to be wrapped

Returns a constant reference to the (up-to-date) CredalNet CPTs.

Return type tbw

credalNet_srcCpt(*CredalNet self*)

Warning: Experimental function - Return type to be wrapped

Returns a constant reference to the (up-to-date) CredalNet CPTs.

Return type tbw

currentNodeType(*CredalNet self*, *int 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 187)

current_bn(*CredalNet self*)

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

Return type *pyAgrum.BayesNet* (page 48)

domainSize(*CredalNet self*, *int id*)

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

Returns The cardinality of the node

Return type int

epsilonMax(*CredalNet self*)

Returns a constant reference to the highest perturbation of the BayesNet provided as input for this CredalNet.

Return type double

epsilonMean(*CredalNet self*)

Returns a constant reference to the average perturbation of the BayesNet provided as input for this CredalNet.

Return type double

epsilonMin(*CredalNet self*)

Returns a constant reference to the lowest perturbation of the BayesNet provided as input for this CredalNet.

Return type double

fillConstraint(*CredalNet self*, *int id*, *int entry*, *Vector lower*, *Vector upper*)
fillConstraint(CredalNet self, int id, Instantiation ins, Vector lower, Vector upper)

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 34)) – 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 !

fillConstraints(*CredalNet self*, *int id*, *Vector lower*, *Vector 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 !

get_CPT_max(*CredalNet self*)

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_CPT_min(*CredalNet self*)

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

hasComputedCPTMinMax(*CredalNet self*)

Returns True this CredalNet has called computeCPTMinMax() to speed-up inference with binary networks and L2U.

Return type bool

idmLearning(*CredalNet self*, *int s=0*, *bool 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.

instantiation(*CredalNet self*, *int 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 34)

intervalToCredal (*CredalNet self*)
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.

intervalToCredalWithFiles (*CredalNet self*)

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.

isSeparatelySpecified (*CredalNet self*)

Returns True if this CredalNet is separately and interval specified, False otherwise.

Return type bool

lagrangeNormalization (*CredalNet self*)
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).

nodeType (*CredalNet self*, *int 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 187)

saveBNsMinMax (*CredalNet self*, *str min_path*, *str 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.

setCPT (*CredalNet self*, *int id*, *int entry*, *vector< vector< double*, *allocator >*, *allocator< vector< double*, *allocator > > > cpt*)
setCPT(CredalNet self, int id, Instantiation ins, vector< vector< double,allocator >,allocator< vector< double,allocator > > > cpt)

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 34)) – 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 !

setCPTs (*CredalNet self*, *int id*, *vector< vector< vector< double*, *allocator >*, *allocator< vector< double*, *allocator > > >*, *allocator< vector< vector< double*, *allocator >*, *allocator< vector< double*, *allocator > > > > > *cpt*)

Warning: (experimental function) - Parameters to be wrapped

Set the vertices of the credal sets (all of the conditionals) of a given node

Parameters

- **id** (*int*) – the NodeId of the node
- **cpt** (*tbw*) – the vertices of every credal set (for each instantiation of the parents)

Warning: DOES not change the BayesNet (s) associated to this credal net !

src_bn (*CredalNet self*)

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

Return type *pyAgrum.BayesNet* (page 48)

8.2 Inference

class pyAgrum.**CNMonteCarloSampling** (*credalNet: pyAgrum.CredalNet*)

Class used for inferences in credal networks with Monte Carlo sampling algorithm.

CNMonteCarloSampling(cn) -> CNMonteCarloSampling

Parameters:

- **cn** (*pyAgrum.CredalNet*) – a credal network

currentTime (*CNMonteCarloSampling self*)

Returns get the current running time in second (double)

Return type double

dynamicExpMax (*CNMonteCarloSampling self*, *str 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 double

dynamicExpMin(*CNMonteCarloSampling self*, *str 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 double

epsilon(*CNMonteCarloSampling self*)

Returns the value of epsilon

Return type double

history(*CNMonteCarloSampling self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

insertEvidenceFile(*CNMonteCarloSampling self*, *str path*)
Insert evidence from file.

Parameters **path** (*str*) – the path to the evidence file.

insertModalsFile(*CNMonteCarloSampling self*, *str path*)
Insert variables modalities from file to compute expectations.

Parameters **path** (*str*) – The path to the modalities file.

makeInference(*CNMonteCarloSampling self*)
Starts the inference.

marginalMax(*CNMonteCarloSampling self*, *int id*)
marginalMax(CNMonteCarloSampling self, str name) -> Vector

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 gum.IndexError – If the node does not belong to the Credal network

marginalMin(*CNMonteCarloSampling self*, *int id*)
marginalMin(CNMonteCarloSampling self, str name) -> Vector

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 gum.IndexError – If the node does not belong to the Credal network

maxIter(*CNMonteCarloSampling self*)

Returns the criterion on number of iterations

Return type int

maxTime(*CNMonteCarloSampling self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*CNMonteCarloSampling self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*CNMonteCarloSampling self*)

Returns the value of the minimal epsilon rate

Return type double

nbrIterations(*CNMonteCarloSampling self*)

Returns the number of iterations

Return type int

periodSize(*CNMonteCarloSampling self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfLowerBound` – If p<1

setEpsilon(*CNMonteCarloSampling self*, *double eps*)

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

Raises `gum.OutOfLowerBound` – If eps<0

setMaxIter(*CNMonteCarloSampling self*, *int max*)

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

Raises `gum.OutOfLowerBound` – If max <= 1

setMaxTime(*CNMonteCarloSampling self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises `gum.OutOfLowerBound` – If timeout<=0.0

setMinEpsilonRate(*CNMonteCarloSampling self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*CNMonteCarloSampling self*, *int p*)

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

Raises `gum.OutOfLowerBound` – If p<1

setRepetitiveInd(*CNMonteCarloSampling self*, *bool flag*)

Parameters **flag** (*bool*) – True if repetitive independence is to be used, false otherwise. Only usefull with dynamic networks.

setVerbosity(*CNMonteCarloSampling self*, *bool v*)

Parameters **v** (*bool*) – verbosity

verbosity(*CNMonteCarloSampling self*)

Returns True if the verbosity is enabled

Return type bool

class pyAgrum.**CNLoopyPropagation**(*cnet: pyAgrum.CredalNet*)

Class used for inferences in credal networks with Loopy Propagation algorithm.

CNLoopyPropagation(cn) -> CNLoopyPropagation

Parameters:

- **cn** (*pyAgrum.CredalNet*) – a Credal network

currentTime(*CNLoopyPropagation self*)

Returns get the current running time in second (double)

Return type double

dynamicExpMax(*CNLoopyPropagation self*, *str 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 double

dynamicExpMin(*CNLoopyPropagation self*, *str 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 double

epsilon(*CNLoopyPropagation self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*CNLoopyPropagation self*)

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.

history(*CNLoopyPropagation self*)

Returns the scheme history

Return type tuple

Raises gum.OperationNotAllowed – If the scheme did not performed or if verbosity is set to false

inferenceType(*CNLoopyPropagation self*, *pyAgrum.credal::CNLoopyPropagation ::InferenceType inft*)

inferenceType(CNLoopyPropagation self) -> pyAgrum.credal::CNLoopyPropagation ::InferenceType

Returns the inference type

Return type int

insertEvidenceFile(*CNLoopyPropagation self*, *str path*)

Insert evidence from file.

Parameters **path** (*str*) – the path to the evidence file.

insertModalsFile(*CNLoopyPropagation self*, *str path*)

Insert variables modalities from file to compute expectations.

Parameters **path** (*str*) – The path to the modalities file.

makeInference(*CNLoopyPropagation self*)

Starts the inference.

marginalMax(*CNLoopyPropagation self*, *int id*)
marginalMax(CNLoopyPropagation self, str name) -> Vector

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 gum.IndexError – If the node does not belong to the Credal network

marginalMin(*CNLoopyPropagation self*, *int id*)
marginalMin(CNLoopyPropagation self, str name) -> Vector

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 gum.IndexError – If the node does not belong to the Credal network

maxIter(*CNLoopyPropagation self*)

Returns the criterion on number of iterations

Return type int

maxTime(*CNLoopyPropagation self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*CNLoopyPropagation self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*CNLoopyPropagation self*)

Returns the value of the minimal epsilon rate

Return type double

nbrIterations(*CNLoopyPropagation self*)

Returns the number of iterations

Return type int

periodSize(*CNLoopyPropagation self*)

Returns the number of samples between 2 stopping

Return type int

Raises gum.OutOfLowerBound – If p<1

saveInference(*CNLoopyPropagation self*, *str path*)
Saves marginals.

Parameters **path** (*str*) – The path to the file to save marginals.

setEpsilon(*CNLoopyPropagation self*, *double eps*)

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

Raises gum.OutOfLowerBound – If eps<0

setMaxIter(*CNLoopyPropagation self*, *int max*)

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

Raises gum.OutOfLowerBound – If max <= 1

setMaxTime(*CNLoopyPropagation self*, *double timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfLowerBound – If timeout<=0.0

setMinEpsilonRate(*CNLoopyPropagation self*, *double rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*CNLoopyPropagation self*, *int p*)

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

Raises gum.OutOfLowerBound – If p<1

setRepetitiveInd(*CNLoopyPropagation self*, *bool flag*)

Parameters **flag** (*bool*) – True if repetitive independence is to be used, false otherwise. Only usefull with dynamic networks.

setVerbosity(*CNLoopyPropagation self*, *bool v*)

Parameters **v** (*bool*) – verbosity

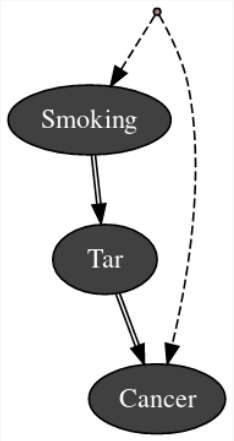
verbosity(*CNLoopyPropagation self*)

Returns True if the verbosity is enabled

Return type bool

CHAPTER 9

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/51-Causality_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: pyAgrum.BayesNet*, *latentVarsDescriptor: Optional[List[Tuple[str, Tuple[str, str]]]] = None*, *keepArcs: bool = 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** – a observational Bayesian network
- **latentVarsDescriptor** – list of couples (<latent variable name>, <list of affected variables' ids>).
- **keepArcs** – 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`

causalBN() → pyAgrum.BayesNet

Returns the causal Bayesian network

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

children(*x: Union[int, str]*) → Set[int]

Parameters **x** – the node

Returns

idFromName(*name: str*) → int

Parameters **name** – the name of the variable

Returns the id of the variable

latentVariablesIds() → Set[int]

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

names() → Dict[int, str]

Returns the map NodeId,Name

observationalBN() → pyAgrum.BayesNet

Returns the observational Bayesian network

parents(*x: Union[int, str]*) → Set[int]

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

Parameters **x** – the node

Returns

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: 'CausalModel'*, *root: ASTtree*, *on: Union(str, NameSet)*, *doing: Union(str, NameSet)*, *knowing: Optional[NameSet] = 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** – the causal model
- **root** – the syntax tree as the root ASTtree
- **on** – the variable or the set of variables of interest
- **doing** – the intervention variables
- **knowing** – the observation variables

cm

return: the causal model

copy() → CausalFormula

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

Returns the new CausalFormula

eval() → pyAgrum.Potential

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

Parameters **bn** – the BN where to infer

Returns

latexQuery(*values: Optional[Dict[str, str]] = None*) → str

Returns a string representing the query compiled by this Formula. If values, the query is annotated with the values in the dictionary.

Parameters **values** – the values to add in the query representation

Returns the string representing the causal query for this CausalFormula

root

return: ASTtree root of the CausalFormula tree

toLatex() → 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: pyAgrum.causal._CausalModel.CausalModel*, *on: Union[str, Set[str]]*, *doing: Union[str, Set[str]]*, *knowing: Optional[Set[str]] = None*, *values: Optional[Dict[str, int]] = None*) → Tuple[pyAgrum.causal._CausalFormula.CausalFormula, pyAgrum.Potential, str]

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** – causal model
- **on** – variable name or variable names set
- **doing** – variable name or variable names set
- **knowing** – variable names set
- **values** – Dictionary

Returns the CausalFormula, the computation, the explanation

pyAgrum.causal.**doCalculusWithObservation**(*cm: pyAgrum.causal._CausalModel.CausalModel, on: str, doing: Set[str], knowing: Optional[Set[str]] = None*) → pyAgrum.causal._CausalFormula.CausalFormula

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** – the variables of interest
- **cm** – the causal model
- **doing** – the interventions
- **knowing** – the observations

Returns the CausalFormula for computing this causal impact

pyAgrum.causal.**identifyingIntervention**(*cm: pyAgrum.causal._CausalModel.CausalModel, Y: Set[str], X: Set[str], P: pyAgrum.causal._doAST.ASTtree = None*) → pyAgrum.causal._doAST.ASTtree

Following Shpitser, Ilya and Judea Pearl. 'Identification of Conditional Interventional Distributions.' UAI2006 and 'Complete Identification Methods for the Causal Hierarchy' JMLR 2008

Parameters

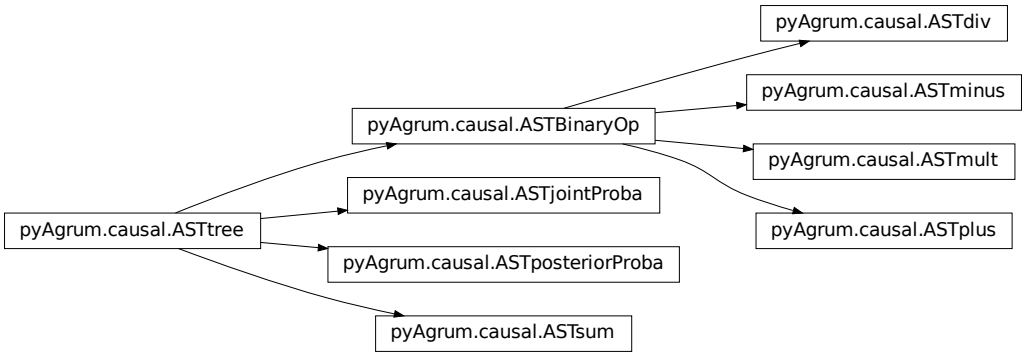
- **cm** – the causal model
- **Y** – The variables of interest (named following the paper)
- **X** – The variable of intervention (named following the paper)
- **P** – The ASTtree representing the calculus in construction

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 203).



9.4.1 Internal node structure

class pyAgrum.causal.**ASTtree**(*type: str*, *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.

copy() → pyAgrum.causal._doAST.ASTtree

Copy an CausalFormula tree

Returns the new causal tree

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str

Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type

return: the type of the node

class pyAgrum.causal.**ASTBinaryOp**(*type: str*, *op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)

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** – left operand
- **op2** – right operand

copy() → pyAgrum.causal._doAST.ASTtree

Copy an CausalFormula tree

Returns the new causal tree

op1

return: the left operand

op2

return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str

Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

9.4.2 Basic Binary Operations

class pyAgrum.causal.**ASTplus**(*op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)
Represents the sum of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTminus**(*op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)
Represents the substraction of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTdiv**(*op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)
Represents the division of 2 causal.ASTtree

Parameters

- **op1** – first operand

- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTmult**(*op1: pyAgrum.causal._doAST.ASTtree, op2: pyAgrum.causal._doAST.ASTtree*)
Represents the multiplication of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

9.4.3 Complex operations

class pyAgrum.causal.**ASTsum**(*var: List[str], term: pyAgrum.causal._doAST.ASTtree*)
Represents a sum over a variable of a causal.ASTtree.

Parameters

- **var** – name of the variable
- **term** – the tree to be evaluated

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

eval(*contextual_bn: pyAgrum.BayesNet*) → pyAgrum.Potential
Evaluation of the sum

Parameters **contextual_bn** – BN where to infer

Returns the value of the sum

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTjointProba**(*varNames: Set[str]*)
Represent a joint probability in the base observational part of the causal.CausalModel

Parameters **varNames** – a set of variable names

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

varNames
return: the set of names of var

class pyAgrum.causal.**ASTposteriorProba**(*bn: pyAgrum.BayesNet, vars: Set[str], knw: Set[str]*)
Represent a conditional probability $P_{bn}(vars|knw)$ that can be computed by an inference in a BN.

Parameters

- **bn** – the pyAgrum:pyAgrum.BayesNet
- **vars** – a set of variable names (in the BN)
- **knw** – a set of variable names (in the BN)

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

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

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

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

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

9.5 Exceptions

class pyAgrum.causal.**HedgeException**(*msg: str, observables: Set[str], gs*)
Represents an hedge exception for a causal query

Parameters

- **msg** – str
- **observables** – NameSet
- **gs** – ???

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

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

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

9.6 Notebook's tools for causality

CHAPTER 10

pyAgrum.skbn documentation

Probabilistic classification in pyAgrum aims to propose a scikit-learn classifier class that can be used in the same codes as sklearn classifiers. Moreover, once the graphical models is built, skbn optimally code the classifier using the smallest set of features needed following the d-separation criterion (Markov Blanket).

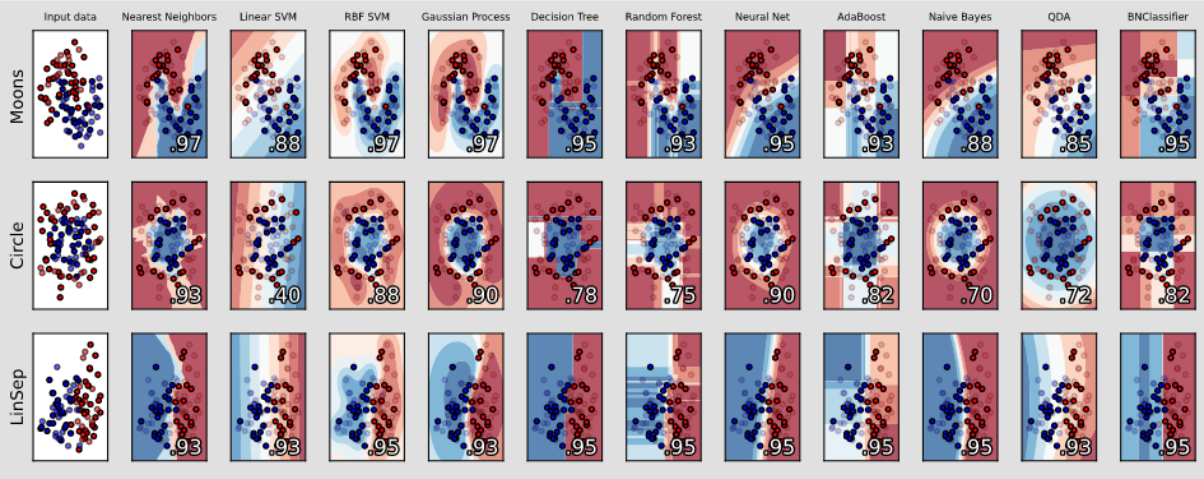


Fig. 1: An example from scikit-learn (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html) where a last column with a BNClassifier has been added flawlessly (see this notebook (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/41-Classifier_Learning.ipynb.html)).

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 sklearn-like classifiers in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/41-Classifier_Learning.ipynb.html) and an example from Kaggle (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/43-Classifier_KaggleTitanic.ipynb.html),
- Notebook on Discretizers in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/42-Classifier_Discretizer_notebook.ipynb.html) useful for sklearn-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 Laplace, BDeu , Dirichlet and None. Note: if using Dirichlet smoothing DirichletCsv cannot be set to none

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=True*, *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 Since 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

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*)

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 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

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 "None" if you do not want to discretize this variable. Possible values are: 'None', 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'

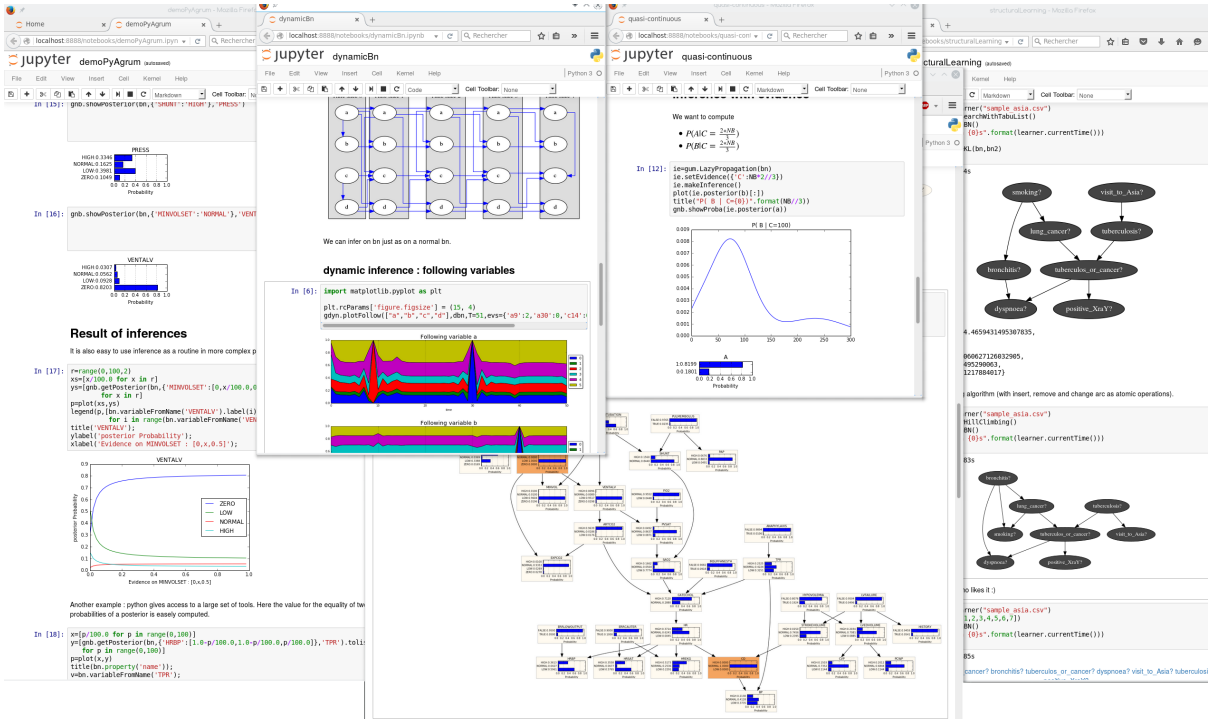
numberOfBins: sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. if the method used is NML, this parameter sets the the maximum number of bins up to which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

CHAPTER 11

pyAgrum.lib.notebook

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



11.1 Visualization of graphical models

11.2 Visualization of Potentials

11.3 Visualization of graphs

11.4 Visualization of approximation algorithm

11.5 Helpers

CHAPTER 12

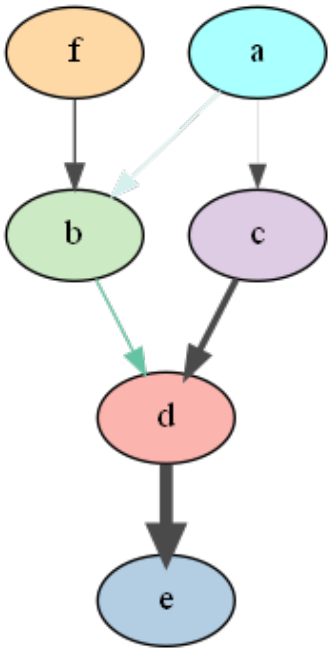
Module bn2graph

A module to graphically display Bayesian networks using pydotplus (https://pypi.org/project/pydotplus/) (and then graphviz (https://graphviz.org/)).

```
import pyAgrum as gum
from pyAgrum.lib.bn2graph import BN2dot

bn = gum.fastBN("a->b->d;a->c->d[3]->e;f->b")
g = BN2dot(bn,
           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})

g.write("bn2graph_test.png", format='png')
```



12.1 Visualization of Potentials

pyAgrum.lib.bn2graph.**proba2histo**(*p*, *scale=1.0*, *util=None*, *txtcolor='Black'*)

compute the representation of an histogram for a mono-dim Potential

Parameters

- **p** (pyAgrum.Potential (page 39)) – the mono-dimensional Potential
- **util** (pyAgrum.Potential (page 39)) – an (optional) secondary Potential (values in labels)

- **txtcolor** (*str*) – color for text

Returns a matplotlib histogram for a Potential p.

Return type matplotlib.Figure

12.2 Visualization of Bayesian networks

pyAgrum.lib.bn2graph.**BN2dot**(*bn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmapNode=None*, *cmapArc=None*, *showMsg=None*)

create a pydotplus representation of the BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the Bayesian network
- **size** (*str*) – size of the rendered graph
- **nodeColor** (*dict*) – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
- **arcWidth** (*dict*) – a arcMap of values to be shown as bold arcs
- **arcColor** (*dict*) – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** (*ColorMap*) – color map to show the vals of Nodes
- **cmapArc** (*ColorMap*) – color map to show the vals of Arcs
- **dag** (pyAgrum.DAG (page 7)) – only shows nodes that have their id in the dag (and not in the whole BN)
- **showMsg** (*dict*) – a nodeMap of values to be shown as tooltip

Returns

Return type the desired representation of the Bayesian network

pyAgrum.lib.bn2graph.**BNinference2dot**(*bn*, *size=None*, *engine=None*, *evs={}*, *targets={}*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmapNode=None*, *cmapArc=None*, *dag=None*)

create a pydotplus representation of an inference in a BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the Bayesian network
- **size** (*str*) – size of the rendered graph
- **engine** (*pyAgrum.Inference*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dict*) – map of evidence
- **targets** (*set*) – set of targets. If targets={} then each node is a target

- **nodeColor** (*dict*) – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
- **arcWidth** (*dict*) – a arcMap of values to be shown as bold arcs
- **arcColor** (*dict*) – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** (*ColorMap*) – color map to show the vals of Nodes
- **cmapArc** (*ColorMap*) – color map to show the vals of Arcs
- **dag** (pyAgrum.DAG (page 7)) – only shows nodes that have their id in the dag (and not in the whole BN)

Returns

Return type the desired representation of the inference

12.3 Hi-level functions

pyAgrum.lib.bn2graph.**dotize**(*aBN*, *name*, *format='pdf'*)
From a bn, creates an image of the BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image
- **format** (*string*) – format in ['pdf','png','fig','jpg','svg']

pyAgrum.lib.bn2graph.**pngize**(*aBN*, *name*)
From a bn, creates a png of the BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image

pyAgrum.lib.bn2graph.**pdfize**(*aBN*, *name*)
From a bn, creates a pdf of the BN

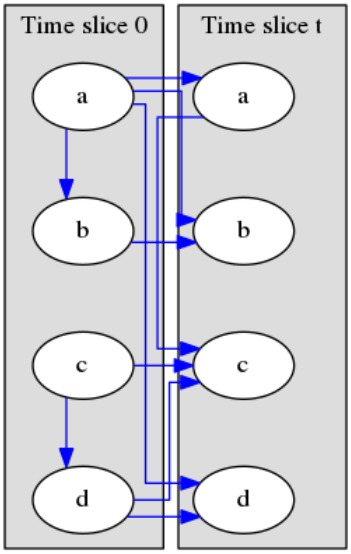
Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image

CHAPTER 13

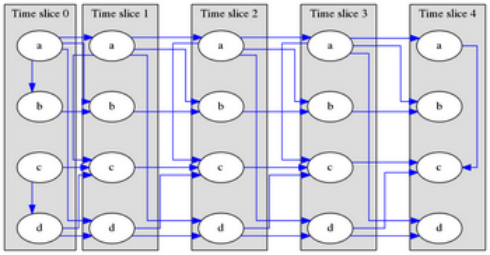
Module dynamic Bayesian network

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



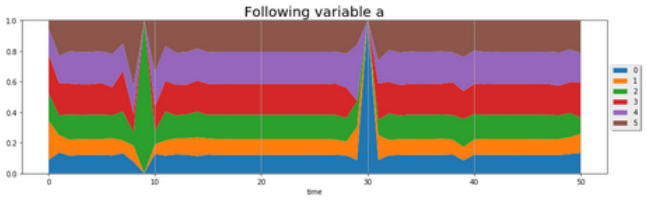

```
T=5

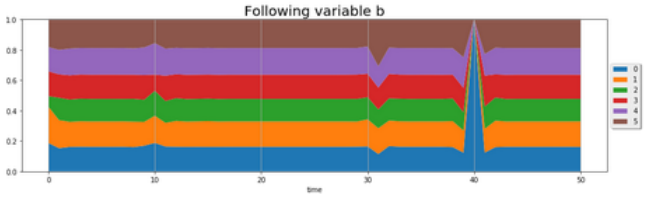
bn=gdyn.unroll2TBN(dbn,T)
gdyn.showTimeSlices(bn,size="10")
```

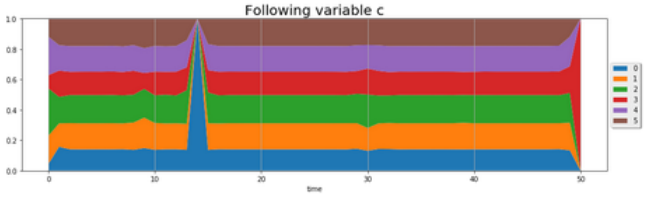


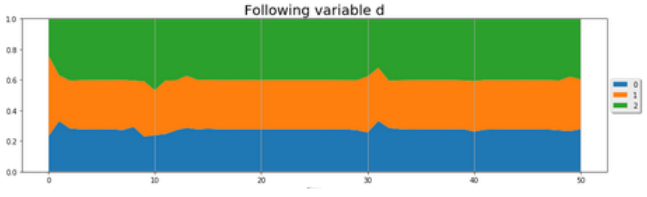

```
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (15, 4)
gdyn.plotFollow(["a","b","c","d"],dbn,T=51,evs={'a9':2,'a
30':0,'c14':0,'b40':0,'c50':3})
```









CHAPTER 14

other pyAgrum.lib modules

14.1 bn2roc

pyAgrum.lib.bn2roc.**module_help**(*exit_value=1*, *message=''*)
defines help viewed if args are not OK on command line, and exit with exit_value

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 48)) – 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=False*, *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 48)) – 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*, *showROC=True*, *showPR=True*, *significant_digits=10*)

Compute the ROC curve and save the result in the folder of the csv file.

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – 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 probabilitie

Returns (pointsROC, seuilROC, pointsPR, seuilPR)

Return type tuple

14.2 bn2scores

pyAgrum.lib.bn2scores.**checkCompatibility**(*bn*, *fields*, *csv_name*)

check if variables of the bn are in the fields

if not : return None if compatibilty : return a list of position for variables in fields

pyAgrum.lib.bn2scores.**computeScores**(*bn_name*, *csv_name*, *visible=False*, *transforme_label=None*)

pyAgrum.lib.bn2scores.**getNumLabel**(*inst*, *i*, *label*, *transforme_label*)

pyAgrum.lib.bn2scores.**lines_count**(*filename*)

count lines in a file

pyAgrum.lib.bn2scores.**module_help**(*exit_value=1*)

defines help viewed if args are not OK on command line, and exit with exit_value

pyAgrum.lib.bn2scores.**stringify**(*s*)

14.3 bn_vs_bn

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 48)) – a BN or a filename for reference
- **name2** (*str or pyAgrum.BayesNet* (page 48)) – 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 parmaeters

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]

pyAgrum.lib.bn_vs_bn.**module_help**(*exit_value=1*)
defines help viewed if args are not OK on command line, and exit with exit_value

14.4 pretty_print

pyAgrum.lib.pretty_print.**bn2txt**(*aBN*)
Representation of all CPTs of a gum.BayesNet

Parameters **aBN** – the bayes net or the name of the file

Returns

pyAgrum.lib.pretty_print.**cpt2txt**(*cpt*, *digits=4*)
string representation of a gum.Potential

Parameters **cpt** – the Potential to represent

Returns the string representation

pyAgrum.lib.pretty_print.**max_length**(*v*)

pyAgrum.lib.pretty_print.**module_help**(*exit_value=1*)
defines help viewed if args are not OK on command line, and exit with exit_value

CHAPTER 15

Functions from pyAgrum

15.1 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 48) *or* pyAgrum.MarkovNet (page 158)) – 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 39) or other)

15.2 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 28) using the default domain size (second argument of the functions).

- with `a[10]`, the variable is a *pyAgrum.RangeVariable* (page 28) using 10 as domain size (from 0 to 9)
- with `a[3,7]`, the variable is a *pyAgrum.RangeVariable* (page 28) using a domainSize from 3 to 7
- with `a[1,3.14,5,6.2]`, the variable is a *pyAgrum.DiscretizedVariable* (page 26) using the given ticks (at least 3 values)
- with `a{top|middle|bottom}`, the variable is a *pyAgrum.LabelizedVariable* (page 23) using the given labels (here : 'top', 'middle' and 'bottom').

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 48)

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 158)

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 170)

15.3 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:
```

(continues on next page)

(continued from previous page)

```
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadBN('./bn.bif',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

pyAgrum.**saveBN**(*bn*, *filename*)
save a BN into a file using the format corresponding to one of the availableWriteBNExts() suffixes.

Parameters

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

15.4 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='%'
```

(continues on next page)

(continued from previous page)

```
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadMN('./bn.uai',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

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

Parameters

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

15.5 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 16

Other functions from aGrUM

16.1 Listeners

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

16.1.1 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
```

16.1.2 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('adding {}:{}'.format(n,
↪s)),
>>>                         whenArcAdded=lambda i,j: print('adding {}->{}'.
↪format(i,j)),
>>>                         whenNodeDeleted=lambda n:print('deleting {}'.
↪format(n)),
>>>                         whenArcDeleted=lambda i,j: print('deleting {}->{}'.
↪format(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
```

16.1.3 ApproximationSchemeListener

16.1.4 DatabaseGenerationListener

16.2 Random functions

pyAgrum.**initRandom**(*unsigned int seed=0*)

Initialize random generator seed.

Parameters **seed** (*int*) – the seed used to initialize the random generator

pyAgrum.**randomProba**()

Returns a random number between 0 and 1 included (i.e. a proba).

Return type double

pyAgrum.**randomDistribution**(*int n*)

Parameters **n** (*int*) – The number of modalities for the ditribution.

Returns

Return type a random discrete distribution.

16.3 OMP functions

pyAgrum.**isOMP**()

Returns True if OMP has been set at compilation, False otherwise

Return type bool

pyAgrum.**setNumberOfThreads**(*unsigned int number*)

To avoid 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

pyAgrum.**getNumberOfLogicalProcessors**()

Returns the number of logical processors

Return type int

pyAgrum.**getMaxNumberOfThreads**()

Returns the max number of threads

Return type int

CHAPTER 17

Exceptions from aGrUM

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

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

Proxy of C++ pyAgrum.DefaultInLabel class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

with_traceback()

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

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

Proxy of C++ pyAgrum.DuplicateElement class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**DuplicateLabel**(**args*)
Proxy of C++ pyAgrum.DuplicateLabel class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**EmptyBSTree**(**args*)
Proxy of C++ pyAgrum.EmptyBSTree class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**EmptySet**(**args*)
Proxy of C++ pyAgrum.EmptySet class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**GumException**(**args*)
Proxy of C++ pyAgrum.Exception class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**FatalError**(**args*)
Proxy of C++ pyAgrum.FatalError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**FormatNotFound**(**args*)
Proxy of C++ pyAgrum.FormatNotFound class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**GraphError**(**args*)
Proxy of C++ pyAgrum.GraphError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**IOError**(**args*)
Proxy of C++ pyAgrum.IOError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**IdError**(**args*)
Proxy of C++ pyAgrum.IdError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidArc**(**args*)
Proxy of C++ pyAgrum.InvalidArc class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidArgument**(**args*)
Proxy of C++ pyAgrum.InvalidArgument class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidArgumentsNumber**(**args*)
Proxy of C++ pyAgrum.InvalidArgumentsNumber class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidDirectedCycle**(**args*)
Proxy of C++ pyAgrum.InvalidDirectedCycle class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidEdge**(**args*)
Proxy of C++ pyAgrum.InvalidEdge class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**InvalidNode**(**args*)
Proxy of C++ pyAgrum.InvalidNode class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**NoChild**(**args*)
Proxy of C++ pyAgrum.NoChild class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**NoNeighbour**(**args*)
Proxy of C++ pyAgrum.NoNeighbour class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**NoParent**(**args*)
Proxy of C++ pyAgrum.NoParent class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**NotFound**(**args*)
Proxy of C++ pyAgrum.NotFound class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**NullElement**(**args*)
Proxy of C++ pyAgrum.NullElement class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**OperationNotAllowed**(**args*)
Proxy of C++ pyAgrum.OperationNotAllowed class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**OutOfBounds**(**args*)
Proxy of C++ pyAgrum.OutOfBounds class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**OutOfLowerBound**(**args*)
Proxy of C++ pyAgrum.OutOfLowerBound class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**OutOfUpperBound**(**args*)
Proxy of C++ pyAgrum.OutOfUpperBound class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**ReferenceError**(**args*)
Proxy of C++ pyAgrum.ReferenceError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**SizeError**(**args*)
Proxy of C++ intError class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**SyntaxError**(**args*)
Proxy of C++ pyAgrum.SyntaxError class.

col(*SyntaxError self*)

Returns the indice of the colonne of the error

Return type int

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

line(*SyntaxError self*)

Returns the indice of the line of the error

Return type int

what(*GumException self*)

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

exception pyAgrum.**UndefinedElement**(**args*)
Proxy of C++ pyAgrum.UndefinedElement class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**UndefinedIteratorKey**(**args*)
Proxy of C++ pyAgrum.UndefinedIteratorKey class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**UndefinedIteratorValue**(**args*)
Proxy of C++ pyAgrum.UndefinedIteratorValue class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

exception pyAgrum.**UnknownLabelInDatabase**(**args*)
Proxy of C++ pyAgrum.UnknownLabelInDatabase class.

errorCallStack(*GumException self*)

Returns the error call stack

Return type str

errorContent(*GumException self*)

Returns the error content

Return type str

errorType(*GumException self*)

Returns the error type

Return type str

what(*GumException self*)

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

CHAPTER 18

Configuration for pyAgrum

Configuration for pyAgrum is centralized in an object `gum.config`, singleton of the class `PyAgrumConfiguration`.

class pyAgrum.**PyAgrumConfiguration**

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/99-Tools_configForPyAgrum.ipynb.html).

Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

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

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 19

Indices and tables

- genindex
- modindex
- search

Index

A

about() (*in module pyAgrum*), 229
abs() (*pyAgrum.Potential method*), 39
add() (*pyAgrum.BayesNet method*), 48
add() (*pyAgrum.InfluenceDiagram method*), 170
add() (*pyAgrum.Instantiation method*), 34
add() (*pyAgrum.MarkovNet method*), 158
add() (*pyAgrum.Potential method*), 39
addAllTargets() (*pyAgrum.GibbsSampling method*), 101
addAllTargets() (*pyAgrum.ImportanceSampling method*), 119
addAllTargets() (*pyAgrum.LazyPropagation method*), 75
addAllTargets() (*pyAgrum.LoopyBeliefPropagation method*), 95
addAllTargets() (*pyAgrum.LoopyGibbsSampling method*), 125
addAllTargets() (*pyAgrum.LoopyImportanceSampling method*), 144
addAllTargets() (*pyAgrum.LoopyMonteCarloSampling method*), 132
addAllTargets() (*pyAgrum.LoopyWeightedSampling method*), 138
addAllTargets() (*pyAgrum.MonteCarloSampling method*), 107
addAllTargets() (*pyAgrum.ShaferShenoyInference method*), 82
addAllTargets() (*pyAgrum.ShaferShenoyMNInference method*), 163
addAllTargets() (*pyAgrum.VariableElimination method*), 88
addAllTargets() (*pyAgrum.WeightedSampling method*), 113
addAMPLITUDE() (*pyAgrum.BayesNet method*), 48
addAND() (*pyAgrum.BayesNet method*), 48
addArc() (*pyAgrum.BayesNet method*), 49
addArc() (*pyAgrum.CredalNet method*), 187
addArc() (*pyAgrum.DAG method*), 7
addArc() (*pyAgrum.DiGraph method*), 5
addArc() (*pyAgrum.InfluenceDiagram method*), 170
addArc() (*pyAgrum.MixedGraph method*), 16
addChanceNode() (*pyAgrum.InfluenceDiagram method*), 170
addCOUNT() (*pyAgrum.BayesNet method*), 49
addDecisionNode() (*pyAgrum.InfluenceDiagram method*), 171
addEdge() (*pyAgrum.CliqueGraph method*), 12
addEdge() (*pyAgrum.MixedGraph method*), 16
addEdge() (*pyAgrum.UndiGraph method*), 9
addEvidence() (*pyAgrum.GibbsSampling method*), 101
addEvidence() (*pyAgrum.ImportanceSampling method*), 119
addEvidence() (*pyAgrum.LazyPropagation method*), 75
addEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 95
addEvidence() (*pyAgrum.LoopyGibbsSampling method*), 125
addEvidence() (*pyAgrum.LoopyImportanceSampling method*), 144
addEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 132
addEvidence() (*pyAgrum.LoopyWeightedSampling method*), 138
addEvidence() (*pyAgrum.MonteCarloSampling method*), 107
addEvidence() (*pyAgrum.ShaferShenoyInference method*), 82
addEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 177
addEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 163
addEvidence() (*pyAgrum.VariableElimination method*), 88

addEvidence() (*pyAgrum.WeightedSampling method*), 113
addEXISTS() (*pyAgrum.BayesNet method*), 49
addFactor() (*pyAgrum.MarkovNet method*), 158
addFORALL() (*pyAgrum.BayesNet method*), 49
addForbiddenArc() (*pyAgrum.BNLearner method*), 150
addJointTarget() (*pyAgrum.LazyPropagation method*), 76
addJointTarget() (*pyAgrum.ShaferShenoyInference method*), 83
addJointTarget() (*pyAgrum.ShaferShenoyMNInference method*), 163
addJointTarget() (*pyAgrum.VariableElimination method*), 89
addLabel() (*pyAgrum.LabelizedVariable method*), 24
addLogit() (*pyAgrum.BayesNet method*), 49
addMandatoryArc() (*pyAgrum.BNLearner method*), 151
addMAX() (*pyAgrum.BayesNet method*), 50
addMEDIAN() (*pyAgrum.BayesNet method*), 50
addMIN() (*pyAgrum.BayesNet method*), 50
addNode() (*pyAgrum.CliqueGraph method*), 12
addNode() (*pyAgrum.DAG method*), 7
addNode() (*pyAgrum.DiGraph method*), 5
addNode() (*pyAgrum.MixedGraph method*), 16
addNode() (*pyAgrum.UndiGraph method*), 10
addNodes() (*pyAgrum.CliqueGraph method*), 12
addNodes() (*pyAgrum.DAG method*), 7
addNodes() (*pyAgrum.DiGraph method*), 5
addNodes() (*pyAgrum.MixedGraph method*), 16
addNodes() (*pyAgrum.UndiGraph method*), 10
addNodeWithId() (*pyAgrum.CliqueGraph method*), 12
addNodeWithId() (*pyAgrum.DAG method*), 7
addNodeWithId() (*pyAgrum.DiGraph method*), 5
addNodeWithId() (*pyAgrum.MixedGraph method*), 16
addNodeWithId() (*pyAgrum.UndiGraph method*), 10
addNoForgettingAssumption() (*pyAgrum.ShaferShenoyLIMIDInference method*), 177
addNoisyAND() (*pyAgrum.BayesNet method*), 50
addNoisyOR() (*pyAgrum.BayesNet method*), 50
addNoisyORCompound() (*pyAgrum.BayesNet method*), 51
addNoisyORNet() (*pyAgrum.BayesNet method*), 51
addOR() (*pyAgrum.BayesNet method*), 51
addPossibleEdge() (*pyAgrum.BNLearner method*), 151
addStructureListener() (*pyAgrum.BayesNet method*), 52
addStructureListener() (*pyAgrum.BayesNetFragment method*), 69
addStructureListener() (*pyAgrum.MarkovNet method*), 158
addSUM() (*pyAgrum.BayesNet method*), 51
addTarget() (*pyAgrum.GibbsSampling method*), 101
addTarget() (*pyAgrum.ImportanceSampling method*), 120
addTarget() (*pyAgrum.LazyPropagation method*), 76
addTarget() (*pyAgrum.LoopyBeliefPropagation method*), 95
addTarget() (*pyAgrum.LoopyGibbsSampling method*), 126
addTarget() (*pyAgrum.LoopyImportanceSampling method*), 144
addTarget() (*pyAgrum.LoopyMonteCarloSampling method*), 132
addTarget() (*pyAgrum.LoopyWeightedSampling method*), 138
addTarget() (*pyAgrum.MonteCarloSampling method*), 107
addTarget() (*pyAgrum.ShaferShenoyInference method*), 83
addTarget() (*pyAgrum.ShaferShenoyMNInference method*), 163
addTarget() (*pyAgrum.VariableElimination method*), 89
addTarget() (*pyAgrum.WeightedSampling method*), 113
addTick() (*pyAgrum.DiscretizedVariable method*), 26
addToClique() (*pyAgrum.CliqueGraph method*), 12
addUtilityNode() (*pyAgrum.InfluenceDiagram method*), 171
addVariable() (*pyAgrum.CredalNet method*), 188
addWeightedArc() (*pyAgrum.BayesNet method*), 52
aggType (*pyAgrum.PRMexplorer attribute*), 181
ancestors() (*pyAgrum.BayesNet method*), 52
ancestors() (*pyAgrum.BayesNetFragment method*), 69
ancestors() (*pyAgrum.DAG method*), 8
ancestors() (*pyAgrum.InfluenceDiagram method*), 172
approximatedBinarization() (*pyAgrum.CredalNet method*), 188
Arc (*class in pyAgrum*), 3
arcs() (*pyAgrum.BayesNet method*), 52
arcs() (*pyAgrum.BayesNetFragment method*), 69
arcs() (*pyAgrum.DAG method*), 8
arcs() (*pyAgrum.DiGraph method*), 5
arcs() (*pyAgrum.EssentialGraph method*), 66
arcs() (*pyAgrum.InfluenceDiagram method*), 172
arcs() (*pyAgrum.MarkovBlanket method*), 68
arcs() (*pyAgrum.MixedGraph method*), 16

argmax() (*pyAgrum.Potential method*), 39
argmin() (*pyAgrum.Potential method*), 39
ASTBinaryOp (*class in pyAgrum.causal*), 203
ASTdiv (*class in pyAgrum.causal*), 204
ASTjointProba (*class in pyAgrum.causal*), 206
ASTminus (*class in pyAgrum.causal*), 204
ASTmult (*class in pyAgrum.causal*), 205
ASTplus (*class in pyAgrum.causal*), 204
ASTposteriorProba (*class in pyAgrum.causal*), 206
ASTsum (*class in pyAgrum.causal*), 205
ASTtree (*class in pyAgrum.causal*), 203
audit() (*pyAgrum.skbn.BNDiscretizer method*), 213
availableBNExts() (*in module pyAgrum*), 231
availableIDExts() (*in module pyAgrum*), 233
availableMNExts() (*in module pyAgrum*), 232

B

BayesNet (*class in pyAgrum*), 48
BayesNetFragment (*class in pyAgrum*), 69
beginTopologyTransformation() (*pyAgrum.BayesNet method*), 52
beginTopologyTransformation() (*pyAgrum.MarkovNet method*), 159
belongs() (*pyAgrum.RangeVariable method*), 29
binaryJoinTree() (*pyAgrum.JunctionTreeGenerator method*), 65
bn (*pyAgrum.causal.ASTposteriorProba attribute*), 206
BN() (*pyAgrum.GibbsSampling method*), 100
BN() (*pyAgrum.ImportanceSampling method*), 119
BN() (*pyAgrum.LazyPropagation method*), 74
BN() (*pyAgrum.LoopyBeliefPropagation method*), 94
BN() (*pyAgrum.LoopyGibbsSampling method*), 125
BN() (*pyAgrum.LoopyImportanceSampling method*), 144
BN() (*pyAgrum.LoopyMonteCarloSampling method*), 131
BN() (*pyAgrum.LoopyWeightedSampling method*), 137
BN() (*pyAgrum.MonteCarloSampling method*), 107
BN() (*pyAgrum.ShaferShenoyInference method*), 81
BN() (*pyAgrum.VariableElimination method*), 88
BN() (*pyAgrum.WeightedSampling method*), 113
BN2dot() (*in module pyAgrum.lib.bn2graph*), 220
BNClassifier (*class in pyAgrum.skbn*), 210
BNDatabaseGenerator (*class in pyAgrum*), 60
BNDiscretizer (*class in pyAgrum.skbn*), 213
BNinference2dot() (*in module pyAgrum.lib.bn2graph*), 220
BNLearner (*class in pyAgrum*), 150
bnToCredal() (*pyAgrum.CredalNet method*), 188
burnIn() (*pyAgrum.GibbsBNdistance method*), 62
burnIn() (*pyAgrum.GibbsSampling method*), 101
burnIn() (*pyAgrum.LoopyGibbsSampling method*), 126

C

causalBN() (*pyAgrum.causal.CausalModel method*), 200
CausalFormula (*class in pyAgrum.causal*), 200
causalImpact() (*in module pyAgrum.causal*), 201
CausalModel (*class in pyAgrum.causal*), 200
chanceNodeSize() (*pyAgrum.InfluenceDiagram method*), 172
changeLabel() (*pyAgrum.LabelizedVariable method*), 24
changePotential() (*pyAgrum.BayesNet method*), 52
changeVariableLabel() (*pyAgrum.BayesNet method*), 53
changeVariableLabel() (*pyAgrum.MarkovNet method*), 159
changeVariableName() (*pyAgrum.BayesNet method*), 53
changeVariableName() (*pyAgrum.InfluenceDiagram method*), 172
changeVariableName() (*pyAgrum.MarkovNet method*), 159
checkConsistency() (*pyAgrum.BayesNetFragment method*), 69
chgEvidence() (*pyAgrum.GibbsSampling method*), 101
chgEvidence() (*pyAgrum.ImportanceSampling method*), 120
chgEvidence() (*pyAgrum.LazyPropagation method*), 76
chgEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 95
chgEvidence() (*pyAgrum.LoopyGibbsSampling method*), 126
chgEvidence() (*pyAgrum.LoopyImportanceSampling method*), 145
chgEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 132
chgEvidence() (*pyAgrum.LoopyWeightedSampling method*), 138
chgEvidence() (*pyAgrum.MonteCarloSampling method*), 108
chgEvidence() (*pyAgrum.ShaferShenoyInference method*), 83
chgEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 177
chgEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 164
chgEvidence() (*pyAgrum.VariableElimination method*), 89
chgEvidence() (*pyAgrum.WeightedSampling method*), 114
chgVal() (*pyAgrum.Instantiation method*), 34

chi2() (*pyAgrum.BNLearner method*), 151
children() (*pyAgrum.BayesNet method*), 53
children() (*pyAgrum.BayesNetFragment method*), 69
children() (*pyAgrum.causal.CausalModel method*), 200
children() (*pyAgrum.DAG method*), 8
children() (*pyAgrum.DiGraph method*), 5
children() (*pyAgrum.EssentialGraph method*), 66
children() (*pyAgrum.InfluenceDiagram method*), 172
children() (*pyAgrum.MarkovBlanket method*), 68
children() (*pyAgrum.MixedGraph method*), 16
classAggregates() (*pyAgrum.PRMexplorer method*), 181
classAttributes() (*pyAgrum.PRMexplorer method*), 181
classDag() (*pyAgrum.PRMexplorer method*), 181
classes() (*pyAgrum.PRMexplorer method*), 182
classImplements() (*pyAgrum.PRMexplorer method*), 181
classParameters() (*pyAgrum.PRMexplorer method*), 182
classReferences() (*pyAgrum.PRMexplorer method*), 182
classSlotChains() (*pyAgrum.PRMexplorer method*), 182
clear() (*pyAgrum.BayesNet method*), 53
clear() (*pyAgrum.CliqueGraph method*), 12
clear() (*pyAgrum.DAG method*), 8
clear() (*pyAgrum.DiGraph method*), 5
clear() (*pyAgrum.InfluenceDiagram method*), 172
clear() (*pyAgrum.Instantiation method*), 35
clear() (*pyAgrum.MarkovNet method*), 159
clear() (*pyAgrum.MixedGraph method*), 16
clear() (*pyAgrum.ShaferShenoyLIMIDInference method*), 177
clear() (*pyAgrum.skbn.BNDiscretizer method*), 213
clear() (*pyAgrum.UndiGraph method*), 10
clearEdges() (*pyAgrum.CliqueGraph method*), 13
clique() (*pyAgrum.CliqueGraph method*), 13
CliqueGraph (*class in pyAgrum*), 12
cm (*pyAgrum.causal.CausalFormula attribute*), 201
CNLoopyPropagation (*class in pyAgrum*), 194
CNMonteCarloSampling (*class in pyAgrum*), 192
col() (*pyAgrum.SyntaxError method*), 249
completeInstantiation() (*pyAgrum.BayesNet method*), 53
completeInstantiation() (*pyAgrum.BayesNetFragment method*), 70
completeInstantiation() (*pyAgrum.InfluenceDiagram method*), 172
completeInstantiation() (*pyAgrum.MarkovNet method*), 159
compute() (*pyAgrum.ExactBNdistance method*), 62
compute() (*pyAgrum.GibbsBNdistance method*), 62
computeCPTMinMax() (*pyAgrum.CredalNet method*), 188
connectedComponents() (*pyAgrum.BayesNet method*), 53
connectedComponents() (*pyAgrum.BayesNetFragment method*), 70
connectedComponents() (*pyAgrum.CliqueGraph method*), 13
connectedComponents() (*pyAgrum.DAG method*), 8
connectedComponents() (*pyAgrum.DiGraph method*), 5
connectedComponents() (*pyAgrum.EssentialGraph method*), 67
connectedComponents() (*pyAgrum.MarkovBlanket method*), 68
connectedComponents() (*pyAgrum.MarkovNet method*), 159
connectedComponents() (*pyAgrum.MixedGraph method*), 16
connectedComponents() (*pyAgrum.UndiGraph method*), 10
container() (*pyAgrum.CliqueGraph method*), 13
containerPath() (*pyAgrum.CliqueGraph method*), 13
contains() (*pyAgrum.Instantiation method*), 35
contains() (*pyAgrum.Potential method*), 39
continueApproximationScheme() (*pyAgrum.GibbsBNdistance method*), 62
copy() (*pyAgrum.causal.ASTBinaryOp method*), 203
copy() (*pyAgrum.causal.ASTdiv method*), 205
copy() (*pyAgrum.causal.ASTjointProba method*), 206
copy() (*pyAgrum.causal.ASTminus method*), 204
copy() (*pyAgrum.causal.ASTmult method*), 205
copy() (*pyAgrum.causal.ASTplus method*), 204
copy() (*pyAgrum.causal.ASTposteriorProba method*), 206
copy() (*pyAgrum.causal.ASTsum method*), 205
copy() (*pyAgrum.causal.ASTtree method*), 203
copy() (*pyAgrum.causal.CausalFormula method*), 201
cpf() (*pyAgrum.PRMexplorer method*), 182
cpt() (*pyAgrum.BayesNet method*), 53
cpt() (*pyAgrum.BayesNetFragment method*), 70
cpt() (*pyAgrum.InfluenceDiagram method*), 172
createVariable() (*pyAgrum.skbn.BNDiscretizer method*), 214
CredalNet (*class in pyAgrum*), 187
credalNet_currentCpt() (*pyAgrum.CredalNet method*), 188
credalNet_srcCpt() (*pyAgrum.CredalNet method*), 188
current_bn() (*pyAgrum.CredalNet method*), 189
currentNodeType() (*pyAgrum.CredalNet method*), 188
currentPosterior() (*pyAgrum.GibbsSampling method*), 102
currentPosterior() (*pyAgrum.ImportanceSampling method*), 120

currentPosterior() (*pyAgrum.LoopyGibbsSampling method*), 126
currentPosterior() (*pyAgrum.LoopyImportanceSampling method*), 145
currentPosterior() (*pyAgrum.LoopyMonteCarloSampling method*), 133
currentPosterior() (*pyAgrum.LoopyWeightedSampling method*), 139
currentPosterior() (*pyAgrum.MonteCarloSampling method*), 108
currentPosterior() (*pyAgrum.WeightedSampling method*), 114
currentTime() (*pyAgrum.BNLearner method*), 151
currentTime() (*pyAgrum.CNLoopyPropagation method*), 195
currentTime() (*pyAgrum.CNMonteCarloSampling method*), 192
currentTime() (*pyAgrum.GibbsBNdistance method*), 62
currentTime() (*pyAgrum.GibbsSampling method*), 102
currentTime() (*pyAgrum.ImportanceSampling method*), 120
currentTime() (*pyAgrum.LoopyBeliefPropagation method*), 96
currentTime() (*pyAgrum.LoopyGibbsSampling method*), 127
currentTime() (*pyAgrum.LoopyImportanceSampling method*), 145
currentTime() (*pyAgrum.LoopyMonteCarloSampling method*), 133
currentTime() (*pyAgrum.LoopyWeightedSampling method*), 139
currentTime() (*pyAgrum.MonteCarloSampling method*), 108
currentTime() (*pyAgrum.WeightedSampling method*), 114

D

DAG (*class in pyAgrum*), 7
dag() (*pyAgrum.BayesNet method*), 54
dag() (*pyAgrum.BayesNetFragment method*), 70
dag() (*pyAgrum.InfluenceDiagram method*), 172
dag() (*pyAgrum.MarkovBlanket method*), 68
database() (*pyAgrum.BNDatabaseGenerator method*), 61
databaseWeight() (*pyAgrum.BNLearner method*), 151
dec() (*pyAgrum.Instantiation method*), 35
decIn() (*pyAgrum.Instantiation method*), 35
decisionNodeSize() (*pyAgrum.InfluenceDiagram method*), 172
decisionOrder() (*pyAgrum.InfluenceDiagram method*), 172
decisionOrderExists() (*pyAgrum.InfluenceDiagram method*), 172
decNotVar() (*pyAgrum.Instantiation method*), 35
decOut() (*pyAgrum.Instantiation method*), 35
decVar() (*pyAgrum.Instantiation method*), 35
DefaultInLabel, 239
descendants() (*pyAgrum.BayesNet method*), 54
descendants() (*pyAgrum.BayesNetFragment method*), 70
descendants() (*pyAgrum.DAG method*), 8
descendants() (*pyAgrum.InfluenceDiagram method*), 173
description() (*pyAgrum.DiscreteVariable method*), 21
description() (*pyAgrum.DiscretizedVariable method*), 26
description() (*pyAgrum.LabelizedVariable method*), 24
description() (*pyAgrum.RangeVariable method*), 29
diff() (*pyAgrum.PyAgrumConfiguration method*), 251
DiGraph (*class in pyAgrum*), 4
dim() (*pyAgrum.BayesNet method*), 54
dim() (*pyAgrum.BayesNetFragment method*), 70
dim() (*pyAgrum.MarkovNet method*), 159
disableEpsilon() (*pyAgrum.GibbsBNdistance method*), 62
disableMaxIter() (*pyAgrum.GibbsBNdistance method*), 62
disableMaxTime() (*pyAgrum.GibbsBNdistance method*), 62
disableMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 63
DiscreteVariable (*class in pyAgrum*), 21
discretizationCAIM() (*pyAgrum.skbn.BNDiscretizer method*), 214
discretizationElbowMethodRotation() (*pyAgrum.skbn.BNDiscretizer method*), 214
discretizationMDLP() (*pyAgrum.skbn.BNDiscretizer method*), 214
discretizationNML() (*pyAgrum.skbn.BNDiscretizer method*), 215
DiscretizedVariable (*class in pyAgrum*), 26
doCalculusWithObservation() (*in module pyAgrum.causal*), 202
domain() (*pyAgrum.DiscreteVariable method*), 21
domain() (*pyAgrum.DiscretizedVariable method*), 26
domain() (*pyAgrum.LabelizedVariable method*), 24
domain() (*pyAgrum.RangeVariable method*), 29
domainSize() (*pyAgrum.BNLearner method*), 151
domainSize() (*pyAgrum.CredalNet method*), 189
domainSize() (*pyAgrum.DiscreteVariable method*), 21

domainSize() (*pyAgrum.DiscretizedVariable method*), 26
domainSize() (*pyAgrum.Instantiation method*), 35
domainSize() (*pyAgrum.LabelizedVariable method*), 24
domainSize() (*pyAgrum.Potential method*), 39
domainSize() (*pyAgrum.RangeVariable method*), 29
dotize() (*in module pyAgrum.lib.bn2graph*), 221
draw() (*pyAgrum.Potential method*), 39
drawSamples() (*pyAgrum.BNDatabaseGenerator method*), 61
DuplicateElement, 239
DuplicateLabel, 240
dynamicExpMax() (*pyAgrum.CNLoopyPropagation method*), 195
dynamicExpMax() (*pyAgrum.CNMonteCarloSampling method*), 192
dynamicExpMin() (*pyAgrum.CNLoopyPropagation method*), 195
dynamicExpMin() (*pyAgrum.CNMonteCarloSampling method*), 193

E

Edge (*class in pyAgrum*), 4
edges() (*pyAgrum.CliqueGraph method*), 13
edges() (*pyAgrum.EssentialGraph method*), 67
edges() (*pyAgrum.MarkovNet method*), 159
edges() (*pyAgrum.MixedGraph method*), 17
edges() (*pyAgrum.UndiGraph method*), 10
eliminationOrder() (*pyAgrum.JunctionTreeGenerator method*), 66
empty() (*pyAgrum.BayesNet method*), 54
empty() (*pyAgrum.BayesNetFragment method*), 70
empty() (*pyAgrum.CliqueGraph method*), 13
empty() (*pyAgrum.DAG method*), 8
empty() (*pyAgrum.DiGraph method*), 5
empty() (*pyAgrum.DiscreteVariable method*), 21
empty() (*pyAgrum.DiscretizedVariable method*), 27
empty() (*pyAgrum.InfluenceDiagram method*), 173
empty() (*pyAgrum.Instantiation method*), 35
empty() (*pyAgrum.LabelizedVariable method*), 24
empty() (*pyAgrum.MarkovNet method*), 159
empty() (*pyAgrum.MixedGraph method*), 17
empty() (*pyAgrum.Potential method*), 39
empty() (*pyAgrum.RangeVariable method*), 29
empty() (*pyAgrum.UndiGraph method*), 10
emptyArcs() (*pyAgrum.DAG method*), 8
emptyArcs() (*pyAgrum.DiGraph method*), 6
emptyArcs() (*pyAgrum.MixedGraph method*), 17
EmptyBSTree, 240
emptyEdges() (*pyAgrum.CliqueGraph method*), 13
emptyEdges() (*pyAgrum.MixedGraph method*), 17
emptyEdges() (*pyAgrum.UndiGraph method*), 10
EmptySet, 240
enableEpsilon() (*pyAgrum.GibbsBNdistance method*), 63
enableMaxIter() (*pyAgrum.GibbsBNdistance method*), 63
enableMaxTime() (*pyAgrum.GibbsBNdistance method*), 63
enableMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 63
end() (*pyAgrum.Instantiation method*), 35
endTopologyTransformation() (*pyAgrum.BayesNet method*), 54
endTopologyTransformation() (*pyAgrum.MarkovNet method*), 160
entropy() (*pyAgrum.Potential method*), 40
epsilon() (*pyAgrum.BNLearner method*), 151
epsilon() (*pyAgrum.CNLoopyPropagation method*), 195
epsilon() (*pyAgrum.CNMonteCarloSampling method*), 193
epsilon() (*pyAgrum.GibbsBNdistance method*), 63
epsilon() (*pyAgrum.GibbsSampling method*), 102
epsilon() (*pyAgrum.ImportanceSampling method*), 120
epsilon() (*pyAgrum.LoopyBeliefPropagation method*), 96
epsilon() (*pyAgrum.LoopyGibbsSampling method*), 127
epsilon() (*pyAgrum.LoopyImportanceSampling method*), 145
epsilon() (*pyAgrum.LoopyMonteCarloSampling method*), 133
epsilon() (*pyAgrum.LoopyWeightedSampling method*), 139
epsilon() (*pyAgrum.MonteCarloSampling method*), 108
epsilon() (*pyAgrum.WeightedSampling method*), 114
epsilonMax() (*pyAgrum.CredalNet method*), 189
epsilonMean() (*pyAgrum.CredalNet method*), 189
epsilonMin() (*pyAgrum.CredalNet method*), 189
erase() (*pyAgrum.BayesNet method*), 54
erase() (*pyAgrum.InfluenceDiagram method*), 173
erase() (*pyAgrum.Instantiation method*), 36
erase() (*pyAgrum.MarkovNet method*), 160
eraseAllEvidence() (*pyAgrum.CNLoopyPropagation method*), 195
eraseAllEvidence() (*pyAgrum.GibbsSampling method*), 102
eraseAllEvidence() (*pyAgrum.ImportanceSampling method*), 121
eraseAllEvidence() (*pyAgrum.LazyPropagation method*), 76
eraseAllEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 96

eraseAllEvidence() (pyAgrum.LoopyGibbsSampling method), 127
eraseAllEvidence() (pyAgrum.LoopyImportanceSampling method), 145
eraseAllEvidence() (pyAgrum.LoopyMonteCarloSampling method), 133
eraseAllEvidence() (pyAgrum.LoopyWeightedSampling method), 139
eraseAllEvidence() (pyAgrum.MonteCarloSampling method), 108
eraseAllEvidence() (pyAgrum.ShaferShenoyInference method), 83
eraseAllEvidence() (pyAgrum.ShaferShenoyLIMIDInference method), 177
eraseAllEvidence() (pyAgrum.ShaferShenoyMNInference method), 164
eraseAllEvidence() (pyAgrum.VariableElimination method), 90
eraseAllEvidence() (pyAgrum.WeightedSampling method), 115
eraseAllJointTargets() (pyAgrum.LazyPropagation method), 76
eraseAllJointTargets() (pyAgrum.ShaferShenoyInference method), 83
eraseAllJointTargets() (pyAgrum.ShaferShenoyMNInference method), 164
eraseAllMarginalTargets() (pyAgrum.LazyPropagation method), 76
eraseAllMarginalTargets() (pyAgrum.ShaferShenoyInference method), 83
eraseAllMarginalTargets() (pyAgrum.ShaferShenoyMNInference method), 164
eraseAllTargets() (pyAgrum.GibbsSampling method), 102
eraseAllTargets() (pyAgrum.ImportanceSampling method), 121
eraseAllTargets() (pyAgrum.LazyPropagation method), 76
eraseAllTargets() (pyAgrum.LoopyBeliefPropagation method), 96
eraseAllTargets() (pyAgrum.LoopyGibbsSampling method), 127
eraseAllTargets() (pyAgrum.LoopyImportanceSampling method), 145
eraseAllTargets() (pyAgrum.LoopyMonteCarloSampling method), 133
eraseAllTargets() (pyAgrum.LoopyWeightedSampling method), 139
eraseAllTargets() (pyAgrum.MonteCarloSampling method), 108
eraseAllTargets() (pyAgrum.ShaferShenoyInference method), 83
eraseAllTargets() (pyAgrum.ShaferShenoyMNInference method), 164
eraseAllTargets() (pyAgrum.VariableElimination method), 90
eraseAllTargets() (pyAgrum.WeightedSampling method), 115
eraseArc() (pyAgrum.BayesNet method), 54
eraseArc() (pyAgrum.DAG method), 8
eraseArc() (pyAgrum.DiGraph method), 6
eraseArc() (pyAgrum.InfluenceDiagram method), 173
eraseArc() (pyAgrum.MixedGraph method), 17
eraseChildren() (pyAgrum.DAG method), 8
eraseChildren() (pyAgrum.DiGraph method), 6
eraseChildren() (pyAgrum.MixedGraph method), 17
eraseEdge() (pyAgrum.CliqueGraph method), 13
eraseEdge() (pyAgrum.MixedGraph method), 17
eraseEdge() (pyAgrum.UndiGraph method), 10
eraseEvidence() (pyAgrum.GibbsSampling method), 102
eraseEvidence() (pyAgrum.ImportanceSampling method), 121
eraseEvidence() (pyAgrum.LazyPropagation method), 77
eraseEvidence() (pyAgrum.LoopyBeliefPropagation method), 96
eraseEvidence() (pyAgrum.LoopyGibbsSampling method), 127
eraseEvidence() (pyAgrum.LoopyImportanceSampling method), 146
eraseEvidence() (pyAgrum.LoopyMonteCarloSampling method), 133
eraseEvidence() (pyAgrum.LoopyWeightedSampling method), 139
eraseEvidence() (pyAgrum.MonteCarloSampling method), 109
eraseEvidence() (pyAgrum.ShaferShenoyInference method), 84
eraseEvidence() (pyAgrum.ShaferShenoyLIMIDInference method), 178
eraseEvidence() (pyA-

grum.ShaferShenoyMNInference method), 164
eraseEvidence() (pyAgrum.VariableElimination method), 90
eraseEvidence() (pyAgrum.WeightedSampling method), 115
eraseFactor() (pyAgrum.MarkovNet method), 160
eraseForbiddenArc() (pyAgrum.BNLearner method), 151
eraseFromClique() (pyAgrum.CliqueGraph method), 14
eraseJointTarget() (pyAgrum.LazyPropagation method), 77
eraseJointTarget() (pyAgrum.ShaferShenoyInference method), 84
eraseJointTarget() (pyAgrum.ShaferShenoyMNInference method), 164
eraseJointTarget() (pyAgrum.VariableElimination method), 90
eraseLabels() (pyAgrum.LabelizedVariable method), 24
eraseMandatoryArc() (pyAgrum.BNLearner method), 152
eraseNeighbours() (pyAgrum.CliqueGraph method), 14
eraseNeighbours() (pyAgrum.MixedGraph method), 17
eraseNeighbours() (pyAgrum.UndiGraph method), 11
eraseNode() (pyAgrum.CliqueGraph method), 14
eraseNode() (pyAgrum.DAG method), 8
eraseNode() (pyAgrum.DiGraph method), 6
eraseNode() (pyAgrum.MixedGraph method), 17
eraseNode() (pyAgrum.UndiGraph method), 11
eraseParents() (pyAgrum.DAG method), 8
eraseParents() (pyAgrum.DiGraph method), 6
eraseParents() (pyAgrum.MixedGraph method), 17
erasePossibleEdge() (pyAgrum.BNLearner method), 152
eraseTarget() (pyAgrum.GibbsSampling method), 102
eraseTarget() (pyAgrum.ImportanceSampling method), 121
eraseTarget() (pyAgrum.LazyPropagation method), 77
eraseTarget() (pyAgrum.LoopyBeliefPropagation method), 96
eraseTarget() (pyAgrum.LoopyGibbsSampling method), 127
eraseTarget() (pyAgrum.LoopyImportanceSampling method), 146
eraseTarget() (pyAgrum.LoopyMonteCarloSampling method), 133
eraseTarget() (pyAgrum.LoopyWeightedSampling method), 140
eraseTarget() (pyAgrum.MonteCarloSampling method), 109
eraseTarget() (pyAgrum.ShaferShenoyInference method), 84
eraseTarget() (pyAgrum.ShaferShenoyMNInference method), 164
eraseTarget() (pyAgrum.VariableElimination method), 90
eraseTarget() (pyAgrum.WeightedSampling method), 115
eraseTicks() (pyAgrum.DiscretizedVariable method), 27
errorCallStack() (pyAgrum.DefaultInLabel method), 239
errorCallStack() (pyAgrum.DuplicateElement method), 239
errorCallStack() (pyAgrum.DuplicateLabel method), 240
errorCallStack() (pyAgrum.EmptyBSTree method), 240
errorCallStack() (pyAgrum.EmptySet method), 240
errorCallStack() (pyAgrum.FatalError method), 241
errorCallStack() (pyAgrum.FormatNotFound method), 241
errorCallStack() (pyAgrum.GraphError method), 242
errorCallStack() (pyAgrum.GumException method), 241
errorCallStack() (pyAgrum.IdError method), 242
errorCallStack() (pyAgrum.InvalidArc method), 243
errorCallStack() (pyAgrum.InvalidArgument method), 243
errorCallStack() (pyAgrum.InvalidArgumentsNumber method), 243
errorCallStack() (pyAgrum.InvalidDirectedCycle method), 244
errorCallStack() (pyAgrum.InvalidEdge method), 244
errorCallStack() (pyAgrum.InvalidNode method), 244
errorCallStack() (pyAgrum.IOError method), 242
errorCallStack() (pyAgrum.NoChild method), 245
errorCallStack() (pyAgrum.NoNeighbour method), 245
errorCallStack() (pyAgrum.NoParent method), 245

errorCallStack() (*pyAgrum.NotFound method*), 246
errorCallStack() (*pyAgrum.NullElement method*), 246
errorCallStack() (*pyAgrum.OperationNotAllowed method*), 246
errorCallStack() (*pyAgrum.OutOfBounds method*), 247
errorCallStack() (*pyAgrum.OutOfLowerBound method*), 247
errorCallStack() (*pyAgrum.OutOfUpperBound method*), 247
errorCallStack() (*pyAgrum.ReferenceError method*), 248
errorCallStack() (*pyAgrum.SizeError method*), 248
errorCallStack() (*pyAgrum.SyntaxError method*), 249
errorCallStack() (*pyAgrum.UndefinedElement method*), 249
errorCallStack() (*pyAgrum.UndefinedIteratorKey method*), 249
errorCallStack() (*pyAgrum.UndefinedIteratorValue method*), 250
errorCallStack() (*pyAgrum.UnknownLabelInDatabase method*), 250
errorContent() (*pyAgrum.DefaultInLabel method*), 239
errorContent() (*pyAgrum.DuplicateElement method*), 239
errorContent() (*pyAgrum.DuplicateLabel method*), 240
errorContent() (*pyAgrum.EmptyBSTree method*), 240
errorContent() (*pyAgrum.EmptySet method*), 240
errorContent() (*pyAgrum.FatalError method*), 241
errorContent() (*pyAgrum.FormatNotFound method*), 241
errorContent() (*pyAgrum.GraphError method*), 242
errorContent() (*pyAgrum.GumException method*), 241
errorContent() (*pyAgrum.IdError method*), 242
errorContent() (*pyAgrum.InvalidArc method*), 243
errorContent() (*pyAgrum.InvalidArgument method*), 243
errorContent() (*pyAgrum.InvalidArgumentsNumber method*), 243
errorContent() (*pyAgrum.InvalidDirectedCycle method*), 244
errorContent() (*pyAgrum.InvalidEdge method*), 244
errorContent() (*pyAgrum.InvalidNode method*), 244
errorContent() (*pyAgrum.IOError method*), 242
errorContent() (*pyAgrum.NoChild method*), 245
errorContent() (*pyAgrum.NoNeighbour method*), 245
errorContent() (*pyAgrum.NoParent method*), 246
errorContent() (*pyAgrum.NotFound method*), 246
errorContent() (*pyAgrum.NullElement method*), 246
errorContent() (*pyAgrum.OperationNotAllowed method*), 247
errorContent() (*pyAgrum.OutOfBounds method*), 247
errorContent() (*pyAgrum.OutOfLowerBound method*), 247
errorContent() (*pyAgrum.OutOfUpperBound method*), 248
errorContent() (*pyAgrum.ReferenceError method*), 248
errorContent() (*pyAgrum.SizeError method*), 248
errorContent() (*pyAgrum.SyntaxError method*), 249
errorContent() (*pyAgrum.UndefinedElement method*), 249
errorContent() (*pyAgrum.UndefinedIteratorKey method*), 249
errorContent() (*pyAgrum.UndefinedIteratorValue method*), 250
errorContent() (*pyAgrum.UnknownLabelInDatabase method*), 250
errorType() (*pyAgrum.DefaultInLabel method*), 239
errorType() (*pyAgrum.DuplicateElement method*), 239
errorType() (*pyAgrum.DuplicateLabel method*), 240
errorType() (*pyAgrum.EmptyBSTree method*), 240
errorType() (*pyAgrum.EmptySet method*), 240
errorType() (*pyAgrum.FatalError method*), 241
errorType() (*pyAgrum.FormatNotFound method*), 241
errorType() (*pyAgrum.GraphError method*), 242
errorType() (*pyAgrum.GumException method*), 241
errorType() (*pyAgrum.IdError method*), 243
errorType() (*pyAgrum.InvalidArc method*), 243
errorType() (*pyAgrum.InvalidArgument method*), 243
errorType() (*pyAgrum.InvalidArgumentsNumber method*), 244
errorType() (*pyAgrum.InvalidDirectedCycle method*), 244
errorType() (*pyAgrum.InvalidEdge method*), 244
errorType() (*pyAgrum.InvalidNode method*), 245
errorType() (*pyAgrum.IOError method*), 242

errorType() (*pyAgrum.NoChild method*), 245
errorType() (*pyAgrum.NoNeighbour method*), 245
errorType() (*pyAgrum.NoParent method*), 246
errorType() (*pyAgrum.NotFound method*), 246
errorType() (*pyAgrum.NullElement method*), 246
errorType() (*pyAgrum.OperationNotAllowed method*), 247
errorType() (*pyAgrum.OutOfBounds method*), 247
errorType() (*pyAgrum.OutOfLowerBound method*), 247
errorType() (*pyAgrum.OutOfUpperBound method*), 248
errorType() (*pyAgrum.ReferenceError method*), 248
errorType() (*pyAgrum.SizeError method*), 248
errorType() (*pyAgrum.SyntaxError method*), 249
errorType() (*pyAgrum.UndefinedElement method*), 249
errorType() (*pyAgrum.UndefinedIteratorKey method*), 250
errorType() (*pyAgrum.UndefinedIteratorValue method*), 250
errorType() (*pyAgrum.UnknownLabelInDatabase method*), 250
EssentialGraph (*class in pyAgrum*), 66
eval() (*pyAgrum.causal.ASTsum method*), 205
eval() (*pyAgrum.causal.CausalFormula method*), 201
evidenceImpact() (*pyAgrum.GibbsSampling method*), 103
evidenceImpact() (*pyAgrum.ImportanceSampling method*), 121
evidenceImpact() (*pyAgrum.LazyPropagation method*), 77
evidenceImpact() (*pyAgrum.LoopyBeliefPropagation method*), 96
evidenceImpact() (*pyAgrum.LoopyGibbsSampling method*), 127
evidenceImpact() (*pyAgrum.LoopyImportanceSampling method*), 146
evidenceImpact() (*pyAgrum.LoopyMonteCarloSampling method*), 134
evidenceImpact() (*pyAgrum.LoopyWeightedSampling method*), 140
evidenceImpact() (*pyAgrum.MonteCarloSampling method*), 109
evidenceImpact() (*pyAgrum.ShaferShenoyInference method*), 84
evidenceImpact() (*pyAgrum.ShaferShenoyMNInference method*), 165
evidenceImpact() (*pyAgrum.VariableElimination method*), 90
evidenceImpact() (*pyAgrum.WeightedSampling method*), 115
evidenceJointImpact() (*pyAgrum.LazyPropagation method*), 77
evidenceJointImpact() (*pyAgrum.ShaferShenoyInference method*), 84
evidenceJointImpact() (*pyAgrum.ShaferShenoyMNInference method*), 165
evidenceJointImpact() (*pyAgrum.VariableElimination method*), 91
evidenceProbability() (*pyAgrum.LazyPropagation method*), 78
evidenceProbability() (*pyAgrum.ShaferShenoyInference method*), 85
evidenceProbability() (*pyAgrum.ShaferShenoyMNInference method*), 165
ExactBNdistance (*class in pyAgrum*), 61
exists() (*pyAgrum.BayesNet method*), 54
exists() (*pyAgrum.BayesNetFragment method*), 70
exists() (*pyAgrum.InfluenceDiagram method*), 173
exists() (*pyAgrum.MarkovNet method*), 160
existsArc() (*pyAgrum.BayesNet method*), 54
existsArc() (*pyAgrum.BayesNetFragment method*), 70
existsArc() (*pyAgrum.DAG method*), 8
existsArc() (*pyAgrum.DiGraph method*), 6
existsArc() (*pyAgrum.InfluenceDiagram method*), 173
existsArc() (*pyAgrum.MixedGraph method*), 18
existsEdge() (*pyAgrum.CliqueGraph method*), 14
existsEdge() (*pyAgrum.MarkovNet method*), 160
existsEdge() (*pyAgrum.MixedGraph method*), 18
existsEdge() (*pyAgrum.UndiGraph method*), 11
existsNode() (*pyAgrum.CliqueGraph method*), 14
existsNode() (*pyAgrum.DAG method*), 8
existsNode() (*pyAgrum.DiGraph method*), 6
existsNode() (*pyAgrum.MixedGraph method*), 18
existsNode() (*pyAgrum.UndiGraph method*), 11
existsPathBetween() (*pyAgrum.InfluenceDiagram method*), 173
extract() (*pyAgrum.Potential method*), 40

F

factor() (*pyAgrum.MarkovNet method*), 160
factors() (*pyAgrum.MarkovNet method*), 160
family() (*pyAgrum.BayesNet method*), 55
family() (*pyAgrum.BayesNetFragment method*), 70
family() (*pyAgrum.InfluenceDiagram method*), 173
fastBN() (*in module pyAgrum*), 230
fastID() (*in module pyAgrum*), 230
fastMN() (*in module pyAgrum*), 230
fastPrototype() (*pyAgrum.BayesNet static method*), 55

fastPrototype() (*pyAgrum.InfluenceDiagram static method*), 173
fastPrototype() (*pyAgrum.MarkovNet static method*), 160
FatalError, 241
fillConstraint() (*pyAgrum.CredalNet method*), 189
fillConstraints() (*pyAgrum.CredalNet method*), 189
fillWith() (*pyAgrum.Potential method*), 40
fillWithFunction() (*pyAgrum.Potential method*), 40
findAll() (*pyAgrum.Potential method*), 41
first() (*pyAgrum.Arc method*), 3
first() (*pyAgrum.Edge method*), 4
fit() (*pyAgrum.skbn.BNClassifier method*), 211
FormatNotFound, 241
fromBN() (*pyAgrum.MarkovNet static method*), 161
fromdict() (*pyAgrum.Instantiation method*), 36
fromTrainedModel() (*pyAgrum.skbn.BNClassifier method*), 211

G

G2() (*pyAgrum.BNLearner method*), 150
generateCPT() (*pyAgrum.BayesNet method*), 55
generateCPTs() (*pyAgrum.BayesNet method*), 55
generateFactor() (*pyAgrum.MarkovNet method*), 161
generateFactors() (*pyAgrum.MarkovNet method*), 161
get() (*pyAgrum.Potential method*), 41
get() (*pyAgrum.PyAgrumConfiguration method*), 251
get_CPT_max() (*pyAgrum.CredalNet method*), 190
get_CPT_min() (*pyAgrum.CredalNet method*), 190
get_params() (*pyAgrum.skbn.BNClassifier method*), 212
getalltheSystems() (*pyAgrum.PRMexplorer method*), 183
getDecisionGraph() (*pyAgrum.InfluenceDiagram method*), 174
getDirectSubClass() (*pyAgrum.PRMexplorer method*), 182
getDirectSubInterfaces() (*pyAgrum.PRMexplorer method*), 182
getDirectSubTypes() (*pyAgrum.PRMexplorer method*), 183
getImplementations() (*pyAgrum.PRMexplorer method*), 183
getLabelMap() (*pyAgrum.PRMexplorer method*), 183
getLabels() (*pyAgrum.PRMexplorer method*), 183
getMaxNumberOfThreads() (*in module pyAgrum*), 237
getNumberOfLogicalProcessors() (*in module pyAgrum*), 237
getPosterior() (*in module pyAgrum*), 229
getSuperClass() (*pyAgrum.PRMexplorer method*), 183
getSuperInterface() (*pyAgrum.PRMexplorer method*), 183
getSuperType() (*pyAgrum.PRMexplorer method*), 183
GibbsBNdistance (*class in pyAgrum*), 62
GibbsSampling (*class in pyAgrum*), 100
graph() (*pyAgrum.MarkovNet method*), 161
GraphError, 242
grep() (*pyAgrum.PyAgrumConfiguration method*), 251
GumException, 241

H

H() (*pyAgrum.GibbsSampling method*), 100
H() (*pyAgrum.ImportanceSampling method*), 119
H() (*pyAgrum.LazyPropagation method*), 75
H() (*pyAgrum.LoopyBeliefPropagation method*), 94
H() (*pyAgrum.LoopyGibbsSampling method*), 125
H() (*pyAgrum.LoopyImportanceSampling method*), 144
H() (*pyAgrum.LoopyMonteCarloSampling method*), 131
H() (*pyAgrum.LoopyWeightedSampling method*), 138
H() (*pyAgrum.MonteCarloSampling method*), 107
H() (*pyAgrum.ShaferShenoyInference method*), 82
H() (*pyAgrum.ShaferShenoyMNInference method*), 162
H() (*pyAgrum.VariableElimination method*), 88
H() (*pyAgrum.WeightedSampling method*), 113
hamming() (*pyAgrum.Instantiation method*), 36
hardEvidenceNodes() (*pyAgrum.GibbsSampling method*), 103
hardEvidenceNodes() (*pyAgrum.ImportanceSampling method*), 121
hardEvidenceNodes() (*pyAgrum.LazyPropagation method*), 78
hardEvidenceNodes() (*pyAgrum.LoopyBeliefPropagation method*), 97
hardEvidenceNodes() (*pyAgrum.LoopyGibbsSampling method*), 127
hardEvidenceNodes() (*pyAgrum.LoopyImportanceSampling method*), 146
hardEvidenceNodes() (*pyAgrum.LoopyMonteCarloSampling method*), 134
hardEvidenceNodes() (*pyAgrum.LoopyWeightedSampling method*), 140
hardEvidenceNodes() (*pyAgrum.MonteCarloSampling method*), 109
hardEvidenceNodes() (*pyAgrum.ShaferShenoyInference method*), 85

hardEvidenceNodes() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
hardEvidenceNodes() (*pyAgrum.ShaferShenoyMNInference method*), 165
hardEvidenceNodes() (*pyAgrum.VariableElimination method*), 91
hardEvidenceNodes() (*pyAgrum.WeightedSampling method*), 115
hasComputedCPTMinMax() (*pyAgrum.CredalNet method*), 190
hasDirectedPath() (*pyAgrum.DAG method*), 8
hasDirectedPath() (*pyAgrum.DiGraph method*), 6
hasDirectedPath() (*pyAgrum.MixedGraph method*), 18
hasEvidence() (*pyAgrum.GibbsSampling method*), 103
hasEvidence() (*pyAgrum.ImportanceSampling method*), 121
hasEvidence() (*pyAgrum.LazyPropagation method*), 78
hasEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 97
hasEvidence() (*pyAgrum.LoopyGibbsSampling method*), 128
hasEvidence() (*pyAgrum.LoopyImportanceSampling method*), 146
hasEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 134
hasEvidence() (*pyAgrum.LoopyWeightedSampling method*), 140
hasEvidence() (*pyAgrum.MonteCarloSampling method*), 109
hasEvidence() (*pyAgrum.ShaferShenoyInference method*), 85
hasEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
hasEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 165
hasEvidence() (*pyAgrum.VariableElimination method*), 91
hasEvidence() (*pyAgrum.WeightedSampling method*), 115
hasHardEvidence() (*pyAgrum.GibbsSampling method*), 103
hasHardEvidence() (*pyAgrum.ImportanceSampling method*), 122
hasHardEvidence() (*pyAgrum.LazyPropagation method*), 78
hasHardEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 97
hasHardEvidence() (*pyAgrum.LoopyGibbsSampling method*), 128
hasHardEvidence() (*pyAgrum.LoopyImportanceSampling method*), 146
hasHardEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 134
hasHardEvidence() (*pyAgrum.LoopyWeightedSampling method*), 140
hasHardEvidence() (*pyAgrum.MonteCarloSampling method*), 109
hasHardEvidence() (*pyAgrum.ShaferShenoyInference method*), 85
hasHardEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
hasHardEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 166
hasHardEvidence() (*pyAgrum.VariableElimination method*), 91
hasHardEvidence() (*pyAgrum.WeightedSampling method*), 116
hasMissingValues() (*pyAgrum.BNLearner method*), 152
hasNoForgettingAssumption() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
hasRunningIntersection() (*pyAgrum.CliqueGraph method*), 14
hasSameStructure() (*pyAgrum.BayesNet method*), 55
hasSameStructure() (*pyAgrum.BayesNetFragment method*), 70
hasSameStructure() (*pyAgrum.InfluenceDiagram method*), 174
hasSameStructure() (*pyAgrum.MarkovBlanket method*), 68
hasSameStructure() (*pyAgrum.MarkovNet method*), 161
hasSoftEvidence() (*pyAgrum.GibbsSampling method*), 103
hasSoftEvidence() (*pyAgrum.ImportanceSampling method*), 122
hasSoftEvidence() (*pyAgrum.LazyPropagation method*), 78
hasSoftEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 97
hasSoftEvidence() (*pyAgrum.LoopyGibbsSampling method*), 128
hasSoftEvidence() (*pyAgrum.LoopyImportanceSampling method*), 147

hasSoftEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 134
hasSoftEvidence() (*pyAgrum.LoopyWeightedSampling method*), 140
hasSoftEvidence() (*pyAgrum.MonteCarloSampling method*), 110
hasSoftEvidence() (*pyAgrum.ShaferShenoyInference method*), 85
hasSoftEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
hasSoftEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 166
hasSoftEvidence() (*pyAgrum.VariableElimination method*), 91
hasSoftEvidence() (*pyAgrum.WeightedSampling method*), 116
hasUndirectedCycle() (*pyAgrum.CliqueGraph method*), 14
hasUndirectedCycle() (*pyAgrum.MixedGraph method*), 18
hasUndirectedCycle() (*pyAgrum.UndiGraph method*), 11
head() (*pyAgrum.Arc method*), 3
HedgeException (*class in pyAgrum.causal*), 206
history() (*pyAgrum.BNLearner method*), 152
history() (*pyAgrum.CNLoopyPropagation method*), 195
history() (*pyAgrum.CNMonteCarloSampling method*), 193
history() (*pyAgrum.GibbsBNdistance method*), 63
history() (*pyAgrum.GibbsSampling method*), 103
history() (*pyAgrum.ImportanceSampling method*), 122
history() (*pyAgrum.LoopyBeliefPropagation method*), 97
history() (*pyAgrum.LoopyGibbsSampling method*), 128
history() (*pyAgrum.LoopyImportanceSampling method*), 147
history() (*pyAgrum.LoopyMonteCarloSampling method*), 135
history() (*pyAgrum.LoopyWeightedSampling method*), 141
history() (*pyAgrum.MonteCarloSampling method*), 110
history() (*pyAgrum.WeightedSampling method*), 116

I

I() (*pyAgrum.LazyPropagation method*), 75
I() (*pyAgrum.ShaferShenoyInference method*), 82
I() (*pyAgrum.ShaferShenoyMNInference method*), 162
identifyingIntervention() (*in module pyAgrum.causal*), 202
IdError, 242
idFromName() (*pyAgrum.BayesNet method*), 56
idFromName() (*pyAgrum.BayesNetFragment method*), 70
idFromName() (*pyAgrum.BNLearner method*), 152
idFromName() (*pyAgrum.causal.CausalModel method*), 200
idFromName() (*pyAgrum.InfluenceDiagram method*), 174
idFromName() (*pyAgrum.MarkovNet method*), 161
idmLearning() (*pyAgrum.CredalNet method*), 190
ids() (*pyAgrum.BayesNet method*), 56
ids() (*pyAgrum.BayesNetFragment method*), 71
ids() (*pyAgrum.InfluenceDiagram method*), 174
ids() (*pyAgrum.MarkovNet method*), 161
ImportanceSampling (*class in pyAgrum*), 119
inc() (*pyAgrum.Instantiation method*), 36
incIn() (*pyAgrum.Instantiation method*), 36
incNotVar() (*pyAgrum.Instantiation method*), 36
incOut() (*pyAgrum.Instantiation method*), 36
incVar() (*pyAgrum.Instantiation method*), 36
index() (*pyAgrum.DiscreteVariable method*), 22
index() (*pyAgrum.DiscretizedVariable method*), 27
index() (*pyAgrum.LabelizedVariable method*), 24
index() (*pyAgrum.RangeVariable method*), 29
inferenceType() (*pyAgrum.CNLoopyPropagation method*), 195
InfluenceDiagram (*class in pyAgrum*), 170
influenceDiagram() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
initApproximationScheme() (*pyAgrum.GibbsBNdistance method*), 63
initRandom() (*in module pyAgrum*), 236
inOverflow() (*pyAgrum.Instantiation method*), 36
insertEvidenceFile() (*pyAgrum.CNLoopyPropagation method*), 195
insertEvidenceFile() (*pyAgrum.CNMonteCarloSampling method*), 193
insertModalsFile() (*pyAgrum.CNLoopyPropagation method*), 195
insertModalsFile() (*pyAgrum.CNMonteCarloSampling method*), 193
installAscendants() (*pyAgrum.BayesNetFragment method*), 71
installCPT() (*pyAgrum.BayesNetFragment method*), 71
installMarginal() (*pyAgrum.BayesNetFragment method*), 71
installNode() (*pyAgrum.BayesNetFragment method*), 71

Instantiation (*class in pyAgrum*), 34
instantiation() (*pyAgrum.CredalNet method*), 190
interAttributes() (*pyAgrum.PRMexplorer method*), 184
interfaces() (*pyAgrum.PRMexplorer method*), 184
interReferences() (*pyAgrum.PRMexplorer method*), 184
intervalToCredal() (*pyAgrum.CredalNet method*), 191
intervalToCredalWithFiles() (*pyAgrum.CredalNet method*), 191
InvalidArc, 243
InvalidArgument, 243
InvalidArgumentsNumber, 243
InvalidDirectedCycle, 244
InvalidEdge, 244
InvalidNode, 244
inverse() (*pyAgrum.Potential method*), 41
IOError, 242
isAttribute() (*pyAgrum.PRMexplorer method*), 184
isChanceNode() (*pyAgrum.InfluenceDiagram method*), 174
isClass() (*pyAgrum.PRMexplorer method*), 184
isDecisionNode() (*pyAgrum.InfluenceDiagram method*), 174
isDrawnAtRandom() (*pyAgrum.GibbsBNdistance method*), 63
isDrawnAtRandom() (*pyAgrum.GibbsSampling method*), 104
isDrawnAtRandom() (*pyAgrum.LoopyGibbsSampling method*), 128
isEnabledEpsilon() (*pyAgrum.GibbsBNdistance method*), 63
isEnabledMaxIter() (*pyAgrum.GibbsBNdistance method*), 63
isEnabledMaxTime() (*pyAgrum.GibbsBNdistance method*), 63
isEnabledMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 63
isIndependent() (*pyAgrum.BayesNet method*), 56
isIndependent() (*pyAgrum.BayesNetFragment method*), 71
isIndependent() (*pyAgrum.DAG method*), 9
isIndependent() (*pyAgrum.InfluenceDiagram method*), 175
isIndependent() (*pyAgrum.MarkovNet method*), 161
isInstalledNode() (*pyAgrum.BayesNetFragment method*), 71
isInterface() (*pyAgrum.PRMexplorer method*), 184
isJoinTree() (*pyAgrum.CliqueGraph method*), 14
isJointTarget() (*pyAgrum.LazyPropagation method*), 78
isJointTarget() (*pyAgrum.ShaferShenoyInference method*), 85
isJointTarget() (*pyAgrum.ShaferShenoyMNInference method*), 166
isJointTarget() (*pyAgrum.VariableElimination method*), 92
isLabel() (*pyAgrum.LabelizedVariable method*), 24
isMutable() (*pyAgrum.Instantiation method*), 36
isNonZeroMap() (*pyAgrum.Potential method*), 41
isOMP() (*in module pyAgrum*), 237
isSeparatelySpecified() (*pyAgrum.CredalNet method*), 191
isSolvable() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
isTarget() (*pyAgrum.GibbsSampling method*), 104
isTarget() (*pyAgrum.ImportanceSampling method*), 122
isTarget() (*pyAgrum.LazyPropagation method*), 79
isTarget() (*pyAgrum.LoopyBeliefPropagation method*), 97
isTarget() (*pyAgrum.LoopyGibbsSampling method*), 128
isTarget() (*pyAgrum.LoopyImportanceSampling method*), 147
isTarget() (*pyAgrum.LoopyMonteCarloSampling method*), 135
isTarget() (*pyAgrum.LoopyWeightedSampling method*), 141
isTarget() (*pyAgrum.MonteCarloSampling method*), 110
isTarget() (*pyAgrum.ShaferShenoyInference method*), 86
isTarget() (*pyAgrum.ShaferShenoyMNInference method*), 166
isTarget() (*pyAgrum.VariableElimination method*), 92
isTarget() (*pyAgrum.WeightedSampling method*), 116
isTick() (*pyAgrum.DiscretizedVariable method*), 27
isType() (*pyAgrum.PRMexplorer method*), 184
isUtilityNode() (*pyAgrum.InfluenceDiagram method*), 175

J

jointMutualInformation() (*pyAgrum.LazyPropagation method*), 79
jointMutualInformation() (*pyAgrum.ShaferShenoyInference method*), 86
jointMutualInformation() (*pyAgrum.ShaferShenoyMNInference method*), 166
jointMutualInformation() (*pyAgrum.VariableElimination method*), 92

jointPosterior() (*pyAgrum.LazyPropagation method*), 79
jointPosterior() (*pyAgrum.ShaferShenoyInference method*), 86
jointPosterior() (*pyAgrum.ShaferShenoyMNInference method*), 167
jointPosterior() (*pyAgrum.VariableElimination method*), 92
jointProbability() (*pyAgrum.BayesNet method*), 56
jointProbability() (*pyAgrum.BayesNetFragment method*), 71
joinTree() (*pyAgrum.LazyPropagation method*), 79
joinTree() (*pyAgrum.ShaferShenoyInference method*), 86
joinTree() (*pyAgrum.ShaferShenoyMNInference method*), 166
jointTargets() (*pyAgrum.LazyPropagation method*), 79
jointTargets() (*pyAgrum.ShaferShenoyInference method*), 86
jointTargets() (*pyAgrum.ShaferShenoyMNInference method*), 167
jointTargets() (*pyAgrum.VariableElimination method*), 92
junctionTree() (*pyAgrum.JunctionTreeGenerator method*), 66
junctionTree() (*pyAgrum.LazyPropagation method*), 79
junctionTree() (*pyAgrum.ShaferShenoyInference method*), 86
junctionTree() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
junctionTree() (*pyAgrum.ShaferShenoyMNInference method*), 167
junctionTree() (*pyAgrum.VariableElimination method*), 92
JunctionTreeGenerator (*class in pyAgrum*), 65

K

KL() (*pyAgrum.Potential method*), 39
knw (*pyAgrum.causal.ASTposteriorProba attribute*), 206

L

label() (*pyAgrum.DiscreteVariable method*), 22
label() (*pyAgrum.DiscretizedVariable method*), 27
label() (*pyAgrum.LabelizedVariable method*), 24
label() (*pyAgrum.RangeVariable method*), 29
LabelizedVariable (*class in pyAgrum*), 23
labels() (*pyAgrum.DiscreteVariable method*), 22
labels() (*pyAgrum.DiscretizedVariable method*), 27
labels() (*pyAgrum.LabelizedVariable method*), 25
labels() (*pyAgrum.RangeVariable method*), 30
lagrangeNormalization() (*pyAgrum.CredalNet method*), 191
latentVariables() (*pyAgrum.BNLearner method*), 152
latentVariablesIds() (*pyAgrum.causal.CausalModel method*), 200
latexQuery() (*pyAgrum.causal.CausalFormula method*), 201
LazyPropagation (*class in pyAgrum*), 74
learnBN() (*pyAgrum.BNLearner method*), 152
learnDAG() (*pyAgrum.BNLearner method*), 153
learnMixedStructure() (*pyAgrum.BNLearner method*), 153
learnParameters() (*pyAgrum.BNLearner method*), 153
line() (*pyAgrum.SyntaxError method*), 249
load() (*pyAgrum.PRMexplorer method*), 184
load() (*pyAgrum.PyAgrumConfiguration method*), 252
loadBIF() (*pyAgrum.BayesNet method*), 56
loadBIFXML() (*pyAgrum.BayesNet method*), 56
loadBIFXML() (*pyAgrum.InfluenceDiagram method*), 175
loadBN() (*in module pyAgrum*), 231
loadDSL() (*pyAgrum.BayesNet method*), 56
loadID() (*in module pyAgrum*), 233
loadMN() (*in module pyAgrum*), 232
loadNET() (*pyAgrum.BayesNet method*), 57
loadO3PRM() (*pyAgrum.BayesNet method*), 57
loadUAI() (*pyAgrum.BayesNet method*), 57
loadUAI() (*pyAgrum.MarkovNet method*), 161
log10DomainSize() (*pyAgrum.BayesNet method*), 57
log10DomainSize() (*pyAgrum.BayesNetFragment method*), 71
log10DomainSize() (*pyAgrum.InfluenceDiagram method*), 175
log10DomainSize() (*pyAgrum.MarkovNet method*), 161
log2() (*pyAgrum.Potential method*), 41
log2JointProbability() (*pyAgrum.BayesNet method*), 57
log2JointProbability() (*pyAgrum.BayesNetFragment method*), 71
log2likelihood() (*pyAgrum.BNDatabaseGenerator method*), 61
logLikelihood() (*pyAgrum.BNLearner method*), 153
loopIn() (*pyAgrum.Potential method*), 41
LoopyBeliefPropagation (*class in pyAgrum*), 94
LoopyGibbsSampling (*class in pyAgrum*), 125
LoopyImportanceSampling (*class in pyAgrum*), 144
LoopyMonteCarloSampling (*class in pyAgrum*),

131
LoopyWeightedSampling (*class in pyAgrum*), 137

M

makeInference() (*pyAgrum.CNLoopyPropagation method*), 195
makeInference() (*pyAgrum.CNMonteCarloSampling method*), 193
makeInference() (*pyAgrum.GibbsSampling method*), 104
makeInference() (*pyAgrum.ImportanceSampling method*), 122
makeInference() (*pyAgrum.LazyPropagation method*), 79
makeInference() (*pyAgrum.LoopyBeliefPropagation method*), 98
makeInference() (*pyAgrum.LoopyGibbsSampling method*), 129
makeInference() (*pyAgrum.LoopyImportanceSampling method*), 147
makeInference() (*pyAgrum.LoopyMonteCarloSampling method*), 135
makeInference() (*pyAgrum.LoopyWeightedSampling method*), 141
makeInference() (*pyAgrum.MonteCarloSampling method*), 110
makeInference() (*pyAgrum.ShaferShenoyInference method*), 86
makeInference() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
makeInference() (*pyAgrum.ShaferShenoyMNInference method*), 167
makeInference() (*pyAgrum.VariableElimination method*), 92
makeInference() (*pyAgrum.WeightedSampling method*), 116
makeInference_() (*pyAgrum.LoopyGibbsSampling method*), 129
makeInference_() (*pyAgrum.LoopyImportanceSampling method*), 147
makeInference_() (*pyAgrum.LoopyMonteCarloSampling method*), 135
makeInference_() (*pyAgrum.LoopyWeightedSampling method*), 141
marginalMax() (*pyAgrum.CNLoopyPropagation method*), 195
marginalMax() (*pyAgrum.CNMonteCarloSampling method*), 193
marginalMin() (*pyAgrum.CNLoopyPropagation method*), 196
marginalMin() (*pyAgrum.CNMonteCarloSampling method*), 193
margMaxIn() (*pyAgrum.Potential method*), 41
margMaxOut() (*pyAgrum.Potential method*), 41
margMinIn() (*pyAgrum.Potential method*), 42
margMinOut() (*pyAgrum.Potential method*), 42
margProdIn() (*pyAgrum.Potential method*), 42
margProdOut() (*pyAgrum.Potential method*), 42
margSumIn() (*pyAgrum.Potential method*), 42
margSumOut() (*pyAgrum.Potential method*), 42
MarkovBlanket (*class in pyAgrum*), 68
MarkovNet (*class in pyAgrum*), 158
max() (*pyAgrum.Potential method*), 42
maxIter() (*pyAgrum.BNLearner method*), 154
maxIter() (*pyAgrum.CNLoopyPropagation method*), 196
maxIter() (*pyAgrum.CNMonteCarloSampling method*), 193
maxIter() (*pyAgrum.GibbsBNdistance method*), 63
maxIter() (*pyAgrum.GibbsSampling method*), 104
maxIter() (*pyAgrum.ImportanceSampling method*), 122
maxIter() (*pyAgrum.LoopyBeliefPropagation method*), 98
maxIter() (*pyAgrum.LoopyGibbsSampling method*), 129
maxIter() (*pyAgrum.LoopyImportanceSampling method*), 147
maxIter() (*pyAgrum.LoopyMonteCarloSampling method*), 135
maxIter() (*pyAgrum.LoopyWeightedSampling method*), 141
maxIter() (*pyAgrum.MonteCarloSampling method*), 110
maxIter() (*pyAgrum.WeightedSampling method*), 116
maxNonOne() (*pyAgrum.Potential method*), 43
maxNonOneParam() (*pyAgrum.BayesNet method*), 57
maxNonOneParam() (*pyAgrum.BayesNetFragment method*), 72
maxNonOneParam() (*pyAgrum.MarkovNet method*), 161
maxParam() (*pyAgrum.BayesNet method*), 58
maxParam() (*pyAgrum.BayesNetFragment method*), 72
maxParam() (*pyAgrum.MarkovNet method*), 161
maxTime() (*pyAgrum.BNLearner method*), 154
maxTime() (*pyAgrum.CNLoopyPropagation method*), 196
maxTime() (*pyAgrum.CNMonteCarloSampling*

method), 194
maxTime() (*pyAgrum.GibbsBNdistance method*), 63
maxTime() (*pyAgrum.GibbsSampling method*), 104
maxTime() (*pyAgrum.ImportanceSampling method*), 122
maxTime() (*pyAgrum.LoopyBeliefPropagation method*), 98
maxTime() (*pyAgrum.LoopyGibbsSampling method*), 129
maxTime() (*pyAgrum.LoopyImportanceSampling method*), 147
maxTime() (*pyAgrum.LoopyMonteCarloSampling method*), 135
maxTime() (*pyAgrum.LoopyWeightedSampling method*), 141
maxTime() (*pyAgrum.MonteCarloSampling method*), 110
maxTime() (*pyAgrum.WeightedSampling method*), 116
maxVal() (*pyAgrum.RangeVariable method*), 30
maxVarDomainSize() (*pyAgrum.BayesNet method*), 58
maxVarDomainSize() (*pyAgrum.BayesNetFragment method*), 72
maxVarDomainSize() (*pyAgrum.MarkovNet method*), 161
meanVar() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
messageApproximationScheme() (*pyAgrum.BNLearner method*), 154
messageApproximationScheme() (*pyAgrum.CNLoopyPropagation method*), 196
messageApproximationScheme() (*pyAgrum.CNMonteCarloSampling method*), 194
messageApproximationScheme() (*pyAgrum.GibbsBNdistance method*), 64
messageApproximationScheme() (*pyAgrum.GibbsSampling method*), 104
messageApproximationScheme() (*pyAgrum.ImportanceSampling method*), 123
messageApproximationScheme() (*pyAgrum.LoopyBeliefPropagation method*), 98
messageApproximationScheme() (*pyAgrum.LoopyGibbsSampling method*), 129
messageApproximationScheme() (*pyAgrum.LoopyImportanceSampling method*), 148
messageApproximationScheme() (*pyAgrum.LoopyMonteCarloSampling method*), 135
messageApproximationScheme() (*pyAgrum.LoopyWeightedSampling method*), 141
messageApproximationScheme() (*pyAgrum.MonteCarloSampling method*), 110
messageApproximationScheme() (*pyAgrum.WeightedSampling method*), 117
MEU() (*pyAgrum.ShaferShenoyLIMIDInference method*), 177
min() (*pyAgrum.Potential method*), 43
minEpsilonRate() (*pyAgrum.BNLearner method*), 154
minEpsilonRate() (*pyAgrum.CNLoopyPropagation method*), 196
minEpsilonRate() (*pyAgrum.CNMonteCarloSampling method*), 194
minEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 64
minEpsilonRate() (*pyAgrum.GibbsSampling method*), 104
minEpsilonRate() (*pyAgrum.ImportanceSampling method*), 123
minEpsilonRate() (*pyAgrum.LoopyBeliefPropagation method*), 98
minEpsilonRate() (*pyAgrum.LoopyGibbsSampling method*), 129
minEpsilonRate() (*pyAgrum.LoopyImportanceSampling method*), 148
minEpsilonRate() (*pyAgrum.LoopyMonteCarloSampling method*), 135
minEpsilonRate() (*pyAgrum.LoopyWeightedSampling method*), 141
minEpsilonRate() (*pyAgrum.MonteCarloSampling method*), 111
minEpsilonRate() (*pyAgrum.WeightedSampling method*), 117
minimalCondSet() (*pyAgrum.BayesNet method*), 58
minimalCondSet() (*pyAgrum.BayesNetFragment method*), 72
minimalCondSet() (*pyAgrum.MarkovNet method*), 161
minNonZero() (*pyAgrum.Potential method*), 43
minNonZeroParam() (*pyAgrum.BayesNet method*), 58
minNonZeroParam() (*pyAgrum.BayesNetFragment method*), 72
minNonZeroParam() (*pyAgrum.MarkovNet method*), 161
minParam() (*pyAgrum.BayesNet method*), 58
minParam() (*pyAgrum.BayesNetFragment method*), 72
minParam() (*pyAgrum.MarkovNet method*), 161
minVal() (*pyAgrum.RangeVariable method*), 30
MixedGraph (*class in pyAgrum*), 16
mixedGraph() (*pyAgrum.EssentialGraph method*), 67
mixedOrientedPath() (*pyAgrum.MixedGraph*

method), 18
mixedUnorientedPath() (*pyAgrum.MixedGraph method*), 18
MN() (*pyAgrum.ShaferShenoyMNInference method*), 163
MonteCarloSampling (*class in pyAgrum*), 107
moralGraph() (*pyAgrum.BayesNet method*), 58
moralGraph() (*pyAgrum.BayesNetFragment method*), 72
moralGraph() (*pyAgrum.DAG method*), 9
moralGraph() (*pyAgrum.InfluenceDiagram method*), 175
moralizedAncestralGraph() (*pyAgrum.BayesNet method*), 58
moralizedAncestralGraph() (*pyAgrum.BayesNetFragment method*), 72
moralizedAncestralGraph() (*pyAgrum.DAG method*), 9
moralizedAncestralGraph() (*pyAgrum.InfluenceDiagram method*), 175

N

name() (*pyAgrum.DiscreteVariable method*), 22
name() (*pyAgrum.DiscretizedVariable method*), 27
name() (*pyAgrum.LabelizedVariable method*), 25
name() (*pyAgrum.RangeVariable method*), 30
nameFromId() (*pyAgrum.BNLearner method*), 154
names() (*pyAgrum.BayesNet method*), 58
names() (*pyAgrum.BayesNetFragment method*), 72
names() (*pyAgrum.BNLearner method*), 154
names() (*pyAgrum.causal.CausalModel method*), 200
names() (*pyAgrum.InfluenceDiagram method*), 175
names() (*pyAgrum.MarkovNet method*), 161
nbCols() (*pyAgrum.BNLearner method*), 154
nbrDim() (*pyAgrum.Instantiation method*), 36
nbrDim() (*pyAgrum.Potential method*), 43
nbrDrawnVar() (*pyAgrum.GibbsBNdistance method*), 64
nbrDrawnVar() (*pyAgrum.GibbsSampling method*), 104
nbrDrawnVar() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrEvidence() (*pyAgrum.GibbsSampling method*), 104
nbrEvidence() (*pyAgrum.ImportanceSampling method*), 123
nbrEvidence() (*pyAgrum.LazyPropagation method*), 79
nbrEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 98
nbrEvidence() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrEvidence() (*pyAgrum.LoopyImportanceSampling method*), 148
nbrEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 135
nbrEvidence() (*pyAgrum.LoopyWeightedSampling method*), 141
nbrEvidence() (*pyAgrum.MonteCarloSampling method*), 111
nbrEvidence() (*pyAgrum.ShaferShenoyInference method*), 86
nbrEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
nbrEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 167
nbrEvidence() (*pyAgrum.VariableElimination method*), 92
nbrEvidence() (*pyAgrum.WeightedSampling method*), 117
nbrHardEvidence() (*pyAgrum.GibbsSampling method*), 104
nbrHardEvidence() (*pyAgrum.ImportanceSampling method*), 123
nbrHardEvidence() (*pyAgrum.LazyPropagation method*), 79
nbrHardEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 98
nbrHardEvidence() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrHardEvidence() (*pyAgrum.LoopyImportanceSampling method*), 148
nbrHardEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 135
nbrHardEvidence() (*pyAgrum.LoopyWeightedSampling method*), 142
nbrHardEvidence() (*pyAgrum.MonteCarloSampling method*), 111
nbrHardEvidence() (*pyAgrum.ShaferShenoyInference method*), 86
nbrHardEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
nbrHardEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 167
nbrHardEvidence() (*pyAgrum.VariableElimination method*), 93
nbrHardEvidence() (*pyAgrum.WeightedSampling method*), 117
nbrIterations() (*pyAgrum.BNLearner method*), 154
nbrIterations() (*pyAgrum.CNLoopyPropagation method*), 196

nbrIterations() (*pyAgrum.CNMonteCarloSampling method*), 194
nbrIterations() (*pyAgrum.GibbsBNdistance method*), 64
nbrIterations() (*pyAgrum.GibbsSampling method*), 105
nbrIterations() (*pyAgrum.ImportanceSampling method*), 123
nbrIterations() (*pyAgrum.LoopyBeliefPropagation method*), 98
nbrIterations() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrIterations() (*pyAgrum.LoopyImportanceSampling method*), 148
nbrIterations() (*pyAgrum.LoopyMonteCarloSampling method*), 135
nbrIterations() (*pyAgrum.LoopyWeightedSampling method*), 142
nbrIterations() (*pyAgrum.MonteCarloSampling method*), 111
nbrIterations() (*pyAgrum.WeightedSampling method*), 117
nbrJointTargets() (*pyAgrum.LazyPropagation method*), 80
nbrJointTargets() (*pyAgrum.ShaferShenoyInference method*), 87
nbrJointTargets() (*pyAgrum.ShaferShenoyMNInference method*), 167
nbRows() (*pyAgrum.BNLearner method*), 154
nbrSoftEvidence() (*pyAgrum.GibbsSampling method*), 105
nbrSoftEvidence() (*pyAgrum.ImportanceSampling method*), 123
nbrSoftEvidence() (*pyAgrum.LazyPropagation method*), 80
nbrSoftEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 98
nbrSoftEvidence() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrSoftEvidence() (*pyAgrum.LoopyImportanceSampling method*), 148
nbrSoftEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 136
nbrSoftEvidence() (*pyAgrum.LoopyWeightedSampling method*), 142
nbrSoftEvidence() (*pyAgrum.MonteCarloSampling method*), 111
nbrSoftEvidence() (*pyAgrum.ShaferShenoyInference method*), 87
nbrSoftEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
nbrSoftEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 167
nbrSoftEvidence() (*pyAgrum.VariableElimination method*), 93
nbrSoftEvidence() (*pyAgrum.WeightedSampling method*), 117
nbrTargets() (*pyAgrum.GibbsSampling method*), 105
nbrTargets() (*pyAgrum.ImportanceSampling method*), 123
nbrTargets() (*pyAgrum.LazyPropagation method*), 80
nbrTargets() (*pyAgrum.LoopyBeliefPropagation method*), 98
nbrTargets() (*pyAgrum.LoopyGibbsSampling method*), 129
nbrTargets() (*pyAgrum.LoopyImportanceSampling method*), 148
nbrTargets() (*pyAgrum.LoopyMonteCarloSampling method*), 136
nbrTargets() (*pyAgrum.LoopyWeightedSampling method*), 142
nbrTargets() (*pyAgrum.MonteCarloSampling method*), 111
nbrTargets() (*pyAgrum.ShaferShenoyInference method*), 87
nbrTargets() (*pyAgrum.ShaferShenoyMNInference method*), 167
nbrTargets() (*pyAgrum.VariableElimination method*), 93
nbrTargets() (*pyAgrum.WeightedSampling method*), 117
neighbours() (*pyAgrum.CliqueGraph method*), 14
neighbours() (*pyAgrum.EssentialGraph method*), 67
neighbours() (*pyAgrum.MarkovNet method*), 161
neighbours() (*pyAgrum.MixedGraph method*), 19
neighbours() (*pyAgrum.UndiGraph method*), 11
new_abs() (*pyAgrum.Potential method*), 43
new_log2() (*pyAgrum.Potential method*), 43
new_sq() (*pyAgrum.Potential method*), 43
newFactory() (*pyAgrum.Potential method*), 43
NoChild, 245
nodeId() (*pyAgrum.BayesNet method*), 58
nodeId() (*pyAgrum.BayesNetFragment method*), 72
nodeId() (*pyAgrum.InfluenceDiagram method*), 175
nodeId() (*pyAgrum.MarkovNet method*), 161
nodes() (*pyAgrum.BayesNet method*), 58

nodes() (*pyAgrum.BayesNetFragment method*), 73
nodes() (*pyAgrum.CliqueGraph method*), 15
nodes() (*pyAgrum.DAG method*), 9
nodes() (*pyAgrum.DiGraph method*), 6
nodes() (*pyAgrum.EssentialGraph method*), 67
nodes() (*pyAgrum.InfluenceDiagram method*), 175
nodes() (*pyAgrum.MarkovBlanket method*), 68
nodes() (*pyAgrum.MarkovNet method*), 161
nodes() (*pyAgrum.MixedGraph method*), 19
nodes() (*pyAgrum.UndiGraph method*), 11
nodes2ConnectedComponent() (*pyAgrum.CliqueGraph method*), 15
nodes2ConnectedComponent() (*pyAgrum.MixedGraph method*), 19
nodes2ConnectedComponent() (*pyAgrum.UndiGraph method*), 11
nodeset() (*pyAgrum.BayesNet method*), 59
nodeset() (*pyAgrum.BayesNetFragment method*), 73
nodeset() (*pyAgrum.InfluenceDiagram method*), 175
nodeset() (*pyAgrum.MarkovNet method*), 161
nodeType() (*pyAgrum.CredalNet method*), 191
noising() (*pyAgrum.Potential method*), 43
NoNeighbour, 245
NoParent, 245
normalize() (*pyAgrum.Potential method*), 43
normalizeAsCPT() (*pyAgrum.Potential method*), 43
NotFound, 246
NullElement, 246
numerical() (*pyAgrum.DiscreteVariable method*), 22
numerical() (*pyAgrum.DiscretizedVariable method*), 27
numerical() (*pyAgrum.LabelizedVariable method*), 25
numerical() (*pyAgrum.RangeVariable method*), 30

O

observationalBN() (*pyAgrum.causal.CausalModel method*), 200
op1 (*pyAgrum.causal.ASTBinaryOp attribute*), 203
op1 (*pyAgrum.causal.ASTdiv attribute*), 205
op1 (*pyAgrum.causal.ASTminus attribute*), 204
op1 (*pyAgrum.causal.ASTmult attribute*), 205
op1 (*pyAgrum.causal.ASTplus attribute*), 204
op2 (*pyAgrum.causal.ASTBinaryOp attribute*), 203
op2 (*pyAgrum.causal.ASTdiv attribute*), 205
op2 (*pyAgrum.causal.ASTminus attribute*), 204
op2 (*pyAgrum.causal.ASTmult attribute*), 205
op2 (*pyAgrum.causal.ASTplus attribute*), 204
OperationNotAllowed, 246
optimalDecision() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
other() (*pyAgrum.Arc method*), 3
other() (*pyAgrum.Edge method*), 4
OutOfBounds, 247
OutOfLowerBound, 247
OutOfUpperBound, 247

P

parents() (*pyAgrum.BayesNet method*), 59
parents() (*pyAgrum.BayesNetFragment method*), 73
parents() (*pyAgrum.causal.CausalModel method*), 200
parents() (*pyAgrum.DAG method*), 9
parents() (*pyAgrum.DiGraph method*), 6
parents() (*pyAgrum.EssentialGraph method*), 67
parents() (*pyAgrum.InfluenceDiagram method*), 175
parents() (*pyAgrum.MarkovBlanket method*), 68
parents() (*pyAgrum.MixedGraph method*), 19
partialUndiGraph() (*pyAgrum.CliqueGraph method*), 15
partialUndiGraph() (*pyAgrum.MixedGraph method*), 19
partialUndiGraph() (*pyAgrum.UndiGraph method*), 11
pdfize() (*in module pyAgrum.lib.bn2graph*), 221
periodSize() (*pyAgrum.BNLearner method*), 154
periodSize() (*pyAgrum.CNLoopyPropagation method*), 196
periodSize() (*pyAgrum.CNMonteCarloSampling method*), 194
periodSize() (*pyAgrum.GibbsBNdistance method*), 64
periodSize() (*pyAgrum.GibbsSampling method*), 105
periodSize() (*pyAgrum.ImportanceSampling method*), 123
periodSize() (*pyAgrum.LoopyBeliefPropagation method*), 99
periodSize() (*pyAgrum.LoopyGibbsSampling method*), 129
periodSize() (*pyAgrum.LoopyImportanceSampling method*), 148
periodSize() (*pyAgrum.LoopyMonteCarloSampling method*), 136
periodSize() (*pyAgrum.LoopyWeightedSampling method*), 142
periodSize() (*pyAgrum.MonteCarloSampling method*), 111
periodSize() (*pyAgrum.WeightedSampling method*), 117
pngize() (*in module pyAgrum.lib.bn2graph*), 221
pos() (*pyAgrum.Instantiation method*), 36
pos() (*pyAgrum.Potential method*), 43
posLabel() (*pyAgrum.LabelizedVariable method*), 25
posterior() (*pyAgrum.GibbsSampling method*), 105

posterior() (*pyAgrum.ImportanceSampling method*), 123
posterior() (*pyAgrum.LazyPropagation method*), 80
posterior() (*pyAgrum.LoopyBeliefPropagation method*), 99
posterior() (*pyAgrum.LoopyGibbsSampling method*), 130
posterior() (*pyAgrum.LoopyImportanceSampling method*), 148
posterior() (*pyAgrum.LoopyMonteCarloSampling method*), 136
posterior() (*pyAgrum.LoopyWeightedSampling method*), 142
posterior() (*pyAgrum.MonteCarloSampling method*), 111
posterior() (*pyAgrum.ShaferShenoyInference method*), 87
posterior() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
posterior() (*pyAgrum.ShaferShenoyMNInference method*), 167
posterior() (*pyAgrum.VariableElimination method*), 93
posterior() (*pyAgrum.WeightedSampling method*), 117
posteriorUtility() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
Potential (*class in pyAgrum*), 39
predict() (*pyAgrum.skbn.BNClassifier method*), 212
predict_proba() (*pyAgrum.skbn.BNClassifier method*), 212
PRMexplorer (*class in pyAgrum*), 181
proba2histo() (*in module pyAgrum.lib.bn2graph*), 219
product() (*pyAgrum.Potential method*), 43
property() (*pyAgrum.BayesNet method*), 59
property() (*pyAgrum.BayesNetFragment method*), 73
property() (*pyAgrum.InfluenceDiagram method*), 176
property() (*pyAgrum.MarkovNet method*), 162
propertyWithDefault() (*pyAgrum.BayesNet method*), 59
propertyWithDefault() (*pyAgrum.BayesNetFragment method*), 73
propertyWithDefault() (*pyAgrum.InfluenceDiagram method*), 176
propertyWithDefault() (*pyAgrum.MarkovNet method*), 162
pseudoCount() (*pyAgrum.BNLearner method*), 154
putFirst() (*pyAgrum.Potential method*), 44
PyAgrumConfiguration (*class in pyAgrum*), 251

R

random() (*pyAgrum.Potential method*), 44
randomCPT() (*pyAgrum.Potential method*), 44
randomDistribution() (*in module pyAgrum*), 236
randomDistribution() (*pyAgrum.Potential method*), 44
randomProba() (*in module pyAgrum*), 236
RangeVariable (*class in pyAgrum*), 28
rawPseudoCount() (*pyAgrum.BNLearner method*), 155
recordWeight() (*pyAgrum.BNLearner method*), 155
reducedGraph() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
reducedLIMID() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
ReferenceError, 248
remainingBurnIn() (*pyAgrum.GibbsBNdistance method*), 64
remove() (*pyAgrum.Potential method*), 44
rend() (*pyAgrum.Instantiation method*), 37
reorder() (*pyAgrum.Instantiation method*), 37
reorganize() (*pyAgrum.Potential method*), 44
reset() (*pyAgrum.PyAgrumConfiguration method*), 252
reverseArc() (*pyAgrum.BayesNet method*), 59
reversePartialOrder() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
root (*pyAgrum.causal.CausalFormula attribute*), 201

S

save() (*pyAgrum.PyAgrumConfiguration method*), 252
saveBIF() (*pyAgrum.BayesNet method*), 59
saveBIFXML() (*pyAgrum.BayesNet method*), 59
saveBIFXML() (*pyAgrum.InfluenceDiagram method*), 176
saveBN() (*in module pyAgrum*), 232
saveBNsMinMax() (*pyAgrum.CredalNet method*), 191
saveDSL() (*pyAgrum.BayesNet method*), 59
saveID() (*in module pyAgrum*), 233
saveInference() (*pyAgrum.CNLoopyPropagation method*), 196
saveMN() (*in module pyAgrum*), 233
saveNET() (*pyAgrum.BayesNet method*), 59
saveO3PRM() (*pyAgrum.BayesNet method*), 59
saveUAI() (*pyAgrum.BayesNet method*), 59
saveUAI() (*pyAgrum.MarkovNet method*), 162
scale() (*pyAgrum.Potential method*), 44
score() (*pyAgrum.skbn.BNClassifier method*), 212
second() (*pyAgrum.Arc method*), 4
second() (*pyAgrum.Edge method*), 4

separator() (*pyAgrum.CliqueGraph method*), 15
set() (*pyAgrum.Potential method*), 44
set() (*pyAgrum.PyAgrumConfiguration method*), 252
set_params() (*pyAgrum.skbn.BNClassifier method*), 212
setAntiTopologicalVarOrder() (*pyAgrum.BNDatabaseGenerator method*), 61
setAprioriWeight() (*pyAgrum.BNLearner method*), 155
setBurnIn() (*pyAgrum.GibbsBNdistance method*), 64
setBurnIn() (*pyAgrum.GibbsSampling method*), 105
setBurnIn() (*pyAgrum.LoopyGibbsSampling method*), 130
setClique() (*pyAgrum.CliqueGraph method*), 15
setCPT() (*pyAgrum.CredalNet method*), 191
setCPTs() (*pyAgrum.CredalNet method*), 192
setDatabaseWeight() (*pyAgrum.BNLearner method*), 155
setDescription() (*pyAgrum.DiscreteVariable method*), 22
setDescription() (*pyAgrum.DiscretizedVariable method*), 27
setDescription() (*pyAgrum.LabelizedVariable method*), 25
setDescription() (*pyAgrum.RangeVariable method*), 30
setDiscretizationParameters() (*pyAgrum.skbn.BNDiscretizer method*), 215
setDrawnAtRandom() (*pyAgrum.GibbsBNdistance method*), 64
setDrawnAtRandom() (*pyAgrum.GibbsSampling method*), 105
setDrawnAtRandom() (*pyAgrum.LoopyGibbsSampling method*), 130
setEpsilon() (*pyAgrum.BNLearner method*), 155
setEpsilon() (*pyAgrum.CNLoopyPropagation method*), 196
setEpsilon() (*pyAgrum.CNMonteCarloSampling method*), 194
setEpsilon() (*pyAgrum.GibbsBNdistance method*), 64
setEpsilon() (*pyAgrum.GibbsSampling method*), 105
setEpsilon() (*pyAgrum.ImportanceSampling method*), 123
setEpsilon() (*pyAgrum.LoopyBeliefPropagation method*), 99
setEpsilon() (*pyAgrum.LoopyGibbsSampling method*), 130
setEpsilon() (*pyAgrum.LoopyImportanceSampling method*), 148
setEpsilon() (*pyAgrum.LoopyMonteCarloSampling method*), 136
setEpsilon() (*pyAgrum.LoopyWeightedSampling method*), 142
setEpsilon() (*pyAgrum.MonteCarloSampling method*), 111
setEpsilon() (*pyAgrum.WeightedSampling method*), 117
setEvidence() (*pyAgrum.GibbsSampling method*), 105
setEvidence() (*pyAgrum.ImportanceSampling method*), 124
setEvidence() (*pyAgrum.LazyPropagation method*), 80
setEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 99
setEvidence() (*pyAgrum.LoopyGibbsSampling method*), 130
setEvidence() (*pyAgrum.LoopyImportanceSampling method*), 148
setEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 136
setEvidence() (*pyAgrum.LoopyWeightedSampling method*), 142
setEvidence() (*pyAgrum.MonteCarloSampling method*), 111
setEvidence() (*pyAgrum.ShaferShenoyInference method*), 87
setEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 178
setEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 168
setEvidence() (*pyAgrum.VariableElimination method*), 93
setEvidence() (*pyAgrum.WeightedSampling method*), 118
setFindBarrenNodesType() (*pyAgrum.LazyPropagation method*), 80
setFindBarrenNodesType() (*pyAgrum.ShaferShenoyInference method*), 87
setFindBarrenNodesType() (*pyAgrum.VariableElimination method*), 93
setFirst() (*pyAgrum.Instantiation method*), 37
setFirstIn() (*pyAgrum.Instantiation method*), 37
setFirstNotVar() (*pyAgrum.Instantiation method*), 37
setFirstOut() (*pyAgrum.Instantiation method*), 37
setFirstVar() (*pyAgrum.Instantiation method*), 37
setInitialDAG() (*pyAgrum.BNLearner method*), 155
setLast() (*pyAgrum.Instantiation method*), 37
setLastIn() (*pyAgrum.Instantiation method*), 37

setLastNotVar() (*pyAgrum.Instantiation method*), 37
setLastOut() (*pyAgrum.Instantiation method*), 37
setLastVar() (*pyAgrum.Instantiation method*), 38
setMaxIndegree() (*pyAgrum.BNLearner method*), 155
setMaxIter() (*pyAgrum.BNLearner method*), 155
setMaxIter() (*pyAgrum.CNLoopyPropagation method*), 197
setMaxIter() (*pyAgrum.CNMonteCarloSampling method*), 194
setMaxIter() (*pyAgrum.GibbsBNdistance method*), 64
setMaxIter() (*pyAgrum.GibbsSampling method*), 106
setMaxIter() (*pyAgrum.ImportanceSampling method*), 124
setMaxIter() (*pyAgrum.LoopyBeliefPropagation method*), 99
setMaxIter() (*pyAgrum.LoopyGibbsSampling method*), 130
setMaxIter() (*pyAgrum.LoopyImportanceSampling method*), 149
setMaxIter() (*pyAgrum.LoopyMonteCarloSampling method*), 136
setMaxIter() (*pyAgrum.LoopyWeightedSampling method*), 142
setMaxIter() (*pyAgrum.MonteCarloSampling method*), 112
setMaxIter() (*pyAgrum.WeightedSampling method*), 118
setMaxTime() (*pyAgrum.BNLearner method*), 155
setMaxTime() (*pyAgrum.CNLoopyPropagation method*), 197
setMaxTime() (*pyAgrum.CNMonteCarloSampling method*), 194
setMaxTime() (*pyAgrum.GibbsBNdistance method*), 64
setMaxTime() (*pyAgrum.GibbsSampling method*), 106
setMaxTime() (*pyAgrum.ImportanceSampling method*), 124
setMaxTime() (*pyAgrum.LoopyBeliefPropagation method*), 99
setMaxTime() (*pyAgrum.LoopyGibbsSampling method*), 130
setMaxTime() (*pyAgrum.LoopyImportanceSampling method*), 149
setMaxTime() (*pyAgrum.LoopyMonteCarloSampling method*), 136
setMaxTime() (*pyAgrum.LoopyWeightedSampling method*), 143
setMaxTime() (*pyAgrum.MonteCarloSampling method*), 112
setMaxTime() (*pyAgrum.WeightedSampling method*), 118
setMaxVal() (*pyAgrum.RangeVariable method*), 30
setMinEpsilonRate() (*pyAgrum.BNLearner method*), 155
setMinEpsilonRate() (*pyAgrum.CNLoopyPropagation method*), 197
setMinEpsilonRate() (*pyAgrum.CNMonteCarloSampling method*), 194
setMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 64
setMinEpsilonRate() (*pyAgrum.GibbsSampling method*), 106
setMinEpsilonRate() (*pyAgrum.ImportanceSampling method*), 124
setMinEpsilonRate() (*pyAgrum.LoopyBeliefPropagation method*), 99
setMinEpsilonRate() (*pyAgrum.LoopyGibbsSampling method*), 130
setMinEpsilonRate() (*pyAgrum.LoopyImportanceSampling method*), 149
setMinEpsilonRate() (*pyAgrum.LoopyMonteCarloSampling method*), 136
setMinEpsilonRate() (*pyAgrum.LoopyWeightedSampling method*), 143
setMinEpsilonRate() (*pyAgrum.MonteCarloSampling method*), 112
setMinEpsilonRate() (*pyAgrum.WeightedSampling method*), 118
setMinVal() (*pyAgrum.RangeVariable method*), 30
setMutable() (*pyAgrum.Instantiation method*), 38
setName() (*pyAgrum.DiscreteVariable method*), 22
setName() (*pyAgrum.DiscretizedVariable method*), 27
setName() (*pyAgrum.LabelizedVariable method*), 25
setName() (*pyAgrum.RangeVariable method*), 30
setNbrDrawnVar() (*pyAgrum.GibbsBNdistance method*), 64
setNbrDrawnVar() (*pyAgrum.GibbsSampling method*), 106
setNbrDrawnVar() (*pyAgrum.LoopyGibbsSampling method*), 130
setNumberOfThreads() (*in module pyAgrum*), 237
setPeriodSize() (*pyAgrum.BNLearner method*), 155
setPeriodSize() (*pyAgrum.CNLoopyPropagation method*), 197
setPeriodSize() (*pyAgrum.CNMonteCarloSampling method*), 194

setPeriodSize() (*pyAgrum.GibbsBNdistance method*), 64
setPeriodSize() (*pyAgrum.GibbsSampling method*), 106
setPeriodSize() (*pyAgrum.ImportanceSampling method*), 124
setPeriodSize() (*pyAgrum.LoopyBeliefPropagation method*), 99
setPeriodSize() (*pyAgrum.LoopyGibbsSampling method*), 130
setPeriodSize() (*pyAgrum.LoopyImportanceSampling method*), 149
setPeriodSize() (*pyAgrum.LoopyMonteCarloSampling method*), 137
setPeriodSize() (*pyAgrum.LoopyWeightedSampling method*), 143
setPeriodSize() (*pyAgrum.MonteCarloSampling method*), 112
setPeriodSize() (*pyAgrum.WeightedSampling method*), 118
setPossibleSkeleton() (*pyAgrum.BNLearner method*), 155
setProperty() (*pyAgrum.BayesNet method*), 59
setProperty() (*pyAgrum.BayesNetFragment method*), 73
setProperty() (*pyAgrum.InfluenceDiagram method*), 176
setProperty() (*pyAgrum.MarkovNet method*), 162
setRandomVarOrder() (*pyAgrum.BNDatabaseGenerator method*), 61
setRecordWeight() (*pyAgrum.BNLearner method*), 155
setRelevantPotentialsFinderType() (*pyAgrum.LazyPropagation method*), 80
setRelevantPotentialsFinderType() (*pyAgrum.VariableElimination method*), 93
setRepetitiveInd() (*pyAgrum.CNLoopyPropagation method*), 197
setRepetitiveInd() (*pyAgrum.CNMonteCarloSampling method*), 194
setSliceOrder() (*pyAgrum.BNLearner method*), 155
setTargets() (*pyAgrum.GibbsSampling method*), 106
setTargets() (*pyAgrum.ImportanceSampling method*), 124
setTargets() (*pyAgrum.LazyPropagation method*), 81
setTargets() (*pyAgrum.LoopyBeliefPropagation method*), 99
setTargets() (*pyAgrum.LoopyGibbsSampling method*), 131
setTargets() (*pyAgrum.LoopyImportanceSampling method*), 149
setTargets() (*pyAgrum.LoopyMonteCarloSampling method*), 137
setTargets() (*pyAgrum.LoopyWeightedSampling method*), 143
setTargets() (*pyAgrum.MonteCarloSampling method*), 112
setTargets() (*pyAgrum.ShaferShenoyInference method*), 87
setTargets() (*pyAgrum.ShaferShenoyMNInference method*), 168
setTargets() (*pyAgrum.VariableElimination method*), 94
setTargets() (*pyAgrum.WeightedSampling method*), 118
setTopologicalVarOrder() (*pyAgrum.BNDatabaseGenerator method*), 61
setTriangulation() (*pyAgrum.LazyPropagation method*), 81
setTriangulation() (*pyAgrum.ShaferShenoyInference method*), 88
setTriangulation() (*pyAgrum.ShaferShenoyMNInference method*), 168
setTriangulation() (*pyAgrum.VariableElimination method*), 94
setVals() (*pyAgrum.Instantiation method*), 38
setVarOrder() (*pyAgrum.BNDatabaseGenerator method*), 61
setVarOrderFromCSV() (*pyAgrum.BNDatabaseGenerator method*), 61
setVerbosity() (*pyAgrum.BNLearner method*), 155
setVerbosity() (*pyAgrum.CNLoopyPropagation method*), 197
setVerbosity() (*pyAgrum.CNMonteCarloSampling method*), 194
setVerbosity() (*pyAgrum.GibbsBNdistance method*), 65
setVerbosity() (*pyAgrum.GibbsSampling method*), 106
setVerbosity() (*pyAgrum.ImportanceSampling method*), 124
setVerbosity() (*pyAgrum.LoopyBeliefPropagation method*), 100
setVerbosity() (*pyAgrum.LoopyGibbsSampling method*), 131

setVerbosity() (pyAgrum.LoopyImportanceSampling method), 149
setVerbosity() (pyAgrum.LoopyMonteCarloSampling method), 137
setVerbosity() (pyAgrum.LoopyWeightedSampling method), 143
setVerbosity() (pyAgrum.MonteCarloSampling method), 112
setVerbosity() (pyAgrum.WeightedSampling method), 118
setVirtualLBPSize() (pyAgrum.LoopyGibbsSampling method), 131
setVirtualLBPSize() (pyAgrum.LoopyImportanceSampling method), 149
setVirtualLBPSize() (pyAgrum.LoopyMonteCarloSampling method), 137
setVirtualLBPSize() (pyAgrum.LoopyWeightedSampling method), 143
ShaferShenoyInference (class in pyAgrum), 81
ShaferShenoyLIMIDInference (class in pyAgrum), 177
ShaferShenoyMNInference (class in pyAgrum), 162
size() (pyAgrum.BayesNet method), 59
size() (pyAgrum.BayesNetFragment method), 73
size() (pyAgrum.CliqueGraph method), 15
size() (pyAgrum.DAG method), 9
size() (pyAgrum.DiGraph method), 7
size() (pyAgrum.EssentialGraph method), 67
size() (pyAgrum.InfluenceDiagram method), 176
size() (pyAgrum.MarkovBlanket method), 68
size() (pyAgrum.MarkovNet method), 162
size() (pyAgrum.MixedGraph method), 19
size() (pyAgrum.UndiGraph method), 11
sizeArcs() (pyAgrum.BayesNet method), 60
sizeArcs() (pyAgrum.BayesNetFragment method), 73
sizeArcs() (pyAgrum.DAG method), 9
sizeArcs() (pyAgrum.DiGraph method), 7
sizeArcs() (pyAgrum.EssentialGraph method), 67
sizeArcs() (pyAgrum.InfluenceDiagram method), 176
sizeArcs() (pyAgrum.MarkovBlanket method), 68
sizeArcs() (pyAgrum.MixedGraph method), 19
sizeEdges() (pyAgrum.CliqueGraph method), 15
sizeEdges() (pyAgrum.EssentialGraph method), 67
sizeEdges() (pyAgrum.MarkovNet method), 162
sizeEdges() (pyAgrum.MixedGraph method), 19
sizeEdges() (pyAgrum.UndiGraph method), 11
SizeError, 248
sizeNodes() (pyAgrum.EssentialGraph method), 67
sizeNodes() (pyAgrum.MarkovBlanket method), 68
skeleton() (pyAgrum.EssentialGraph method), 67
smallestFactorFromNode() (pyAgrum.MarkovNet method), 162
softEvidenceNodes() (pyAgrum.GibbsSampling method), 106
softEvidenceNodes() (pyAgrum.ImportanceSampling method), 124
softEvidenceNodes() (pyAgrum.LazyPropagation method), 81
softEvidenceNodes() (pyAgrum.LoopyBeliefPropagation method), 100
softEvidenceNodes() (pyAgrum.LoopyGibbsSampling method), 131
softEvidenceNodes() (pyAgrum.LoopyImportanceSampling method), 149
softEvidenceNodes() (pyAgrum.LoopyMonteCarloSampling method), 137
softEvidenceNodes() (pyAgrum.LoopyWeightedSampling method), 143
softEvidenceNodes() (pyAgrum.MonteCarloSampling method), 112
softEvidenceNodes() (pyAgrum.ShaferShenoyInference method), 88
softEvidenceNodes() (pyAgrum.ShaferShenoyLIMIDInference method), 179
softEvidenceNodes() (pyAgrum.ShaferShenoyMNInference method), 168
softEvidenceNodes() (pyAgrum.VariableElimination method), 94
softEvidenceNodes() (pyAgrum.WeightedSampling method), 118
sq() (pyAgrum.Potential method), 44
src_bn() (pyAgrum.CredalNet method), 192
startOfPeriod() (pyAgrum.GibbsBNdistance method), 65
stateApproximationScheme() (pyAgrum.GibbsBNdistance method), 65
stopApproximationScheme() (pyAgrum.GibbsBNdistance method), 65
sum() (pyAgrum.Potential method), 44
SyntaxError, 248

T

tail() (pyAgrum.Arc method), 4
targets() (pyAgrum.GibbsSampling method), 106
targets() (pyAgrum.ImportanceSampling method), 124
targets() (pyAgrum.LazyPropagation method), 81

targets() (*pyAgrum.LoopyBeliefPropagation method*), 100
targets() (*pyAgrum.LoopyGibbsSampling method*), 131
targets() (*pyAgrum.LoopyImportanceSampling method*), 149
targets() (*pyAgrum.LoopyMonteCarloSampling method*), 137
targets() (*pyAgrum.LoopyWeightedSampling method*), 143
targets() (*pyAgrum.MonteCarloSampling method*), 112
targets() (*pyAgrum.ShaferShenoyInference method*), 88
targets() (*pyAgrum.ShaferShenoyMNInference method*), 168
targets() (*pyAgrum.VariableElimination method*), 94
targets() (*pyAgrum.WeightedSampling method*), 118
tick() (*pyAgrum.DiscretizedVariable method*), 27
ticks() (*pyAgrum.DiscretizedVariable method*), 28
toarray() (*pyAgrum.Potential method*), 44
toBN() (*pyAgrum.BayesNetFragment method*), 73
toCSV() (*pyAgrum.BNDatabaseGenerator method*), 61
toDatabaseTable() (*pyAgrum.BNDatabaseGenerator method*), 61
todict() (*pyAgrum.Instantiation method*), 38
toDiscretizedVar() (*pyAgrum.DiscreteVariable method*), 22
toDiscretizedVar() (*pyAgrum.DiscretizedVariable method*), 28
toDiscretizedVar() (*pyAgrum.LabelizedVariable method*), 25
toDiscretizedVar() (*pyAgrum.RangeVariable method*), 30
toDot() (*pyAgrum.BayesNet method*), 60
toDot() (*pyAgrum.BayesNetFragment method*), 73
toDot() (*pyAgrum.CliqueGraph method*), 15
toDot() (*pyAgrum.DAG method*), 9
toDot() (*pyAgrum.DiGraph method*), 7
toDot() (*pyAgrum.EssentialGraph method*), 67
toDot() (*pyAgrum.InfluenceDiagram method*), 176
toDot() (*pyAgrum.MarkovBlanket method*), 69
toDot() (*pyAgrum.MarkovNet method*), 162
toDot() (*pyAgrum.MixedGraph method*), 19
toDot() (*pyAgrum.UndiGraph method*), 12
toDotAsFactorGraph() (*pyAgrum.MarkovNet method*), 162
toDotWithNames() (*pyAgrum.CliqueGraph method*), 15
toLabelizedVar() (*pyAgrum.DiscreteVariable method*), 22
toLabelizedVar() (*pyAgrum.DiscretizedVariable method*), 28
toLabelizedVar() (*pyAgrum.LabelizedVariable method*), 25
toLabelizedVar() (*pyAgrum.RangeVariable method*), 31
toLatex() (*pyAgrum.causal.ASTBinaryOp method*), 203
toLatex() (*pyAgrum.causal.ASTdiv method*), 205
toLatex() (*pyAgrum.causal.ASTjointProba method*), 206
toLatex() (*pyAgrum.causal.ASTminus method*), 204
toLatex() (*pyAgrum.causal.ASTmult method*), 205
toLatex() (*pyAgrum.causal.ASTplus method*), 204
toLatex() (*pyAgrum.causal.ASTposteriorProba method*), 206
toLatex() (*pyAgrum.causal.ASTsum method*), 206
toLatex() (*pyAgrum.causal.ASTtree method*), 203
toLatex() (*pyAgrum.causal.CausalFormula method*), 201
tolist() (*pyAgrum.Potential method*), 45
topologicalOrder() (*pyAgrum.BayesNet method*), 60
topologicalOrder() (*pyAgrum.BayesNetFragment method*), 73
topologicalOrder() (*pyAgrum.DAG method*), 9
topologicalOrder() (*pyAgrum.DiGraph method*), 7
topologicalOrder() (*pyAgrum.InfluenceDiagram method*), 176
topologicalOrder() (*pyAgrum.MixedGraph method*), 19
toRangeVar() (*pyAgrum.DiscreteVariable method*), 22
toRangeVar() (*pyAgrum.DiscretizedVariable method*), 28
toRangeVar() (*pyAgrum.LabelizedVariable method*), 25
toRangeVar() (*pyAgrum.RangeVariable method*), 31
toStringWithDescription() (*pyAgrum.DiscreteVariable method*), 22
toStringWithDescription() (*pyAgrum.DiscretizedVariable method*), 28
toStringWithDescription() (*pyAgrum.LabelizedVariable method*), 25
toStringWithDescription() (*pyAgrum.RangeVariable method*), 31
translate() (*pyAgrum.Potential method*), 45
type (*pyAgrum.causal.ASTBinaryOp attribute*), 203
type (*pyAgrum.causal.ASTdiv attribute*), 205
type (*pyAgrum.causal.ASTjointProba attribute*), 206
type (*pyAgrum.causal.ASTminus attribute*), 204
type (*pyAgrum.causal.ASTmult attribute*), 205
type (*pyAgrum.causal.ASTplus attribute*), 204
type (*pyAgrum.causal.ASTposteriorProba attribute*), 206
type (*pyAgrum.causal.ASTsum attribute*), 206
type (*pyAgrum.causal.ASTtree attribute*), 203
types() (*pyAgrum.PRMexplorer method*), 185

U

UndefinedElement, 249
UndefinedIteratorKey, 249
UndefinedIteratorValue, 250
UndiGraph (*class in pyAgrum*), 9
UnidentifiableException (*class in pyAgrum.causal*), 207
uninstallCPT() (*pyAgrum.BayesNetFragment method*), 73
uninstallNode() (*pyAgrum.BayesNetFragment method*), 73
UnknownLabelInDatabase, 250
unsetEnd() (*pyAgrum.Instantiation method*), 38
unsetOverflow() (*pyAgrum.Instantiation method*), 38
updateApproximationScheme() (*pyAgrum.GibbsBNdistance method*), 65
updateEvidence() (*pyAgrum.GibbsSampling method*), 106
updateEvidence() (*pyAgrum.ImportanceSampling method*), 124
updateEvidence() (*pyAgrum.LazyPropagation method*), 81
updateEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 100
updateEvidence() (*pyAgrum.LoopyGibbsSampling method*), 131
updateEvidence() (*pyAgrum.LoopyImportanceSampling method*), 149
updateEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 137
updateEvidence() (*pyAgrum.LoopyWeightedSampling method*), 143
updateEvidence() (*pyAgrum.MonteCarloSampling method*), 112
updateEvidence() (*pyAgrum.ShaferShenoyInference method*), 88
updateEvidence() (*pyAgrum.ShaferShenoyLIMIDInference method*), 179
updateEvidence() (*pyAgrum.ShaferShenoyMNInference method*), 168
updateEvidence() (*pyAgrum.VariableElimination method*), 94
updateEvidence() (*pyAgrum.WeightedSampling method*), 118
use3off2() (*pyAgrum.BNLearner method*), 155
useAprioriBDeu() (*pyAgrum.BNLearner method*), 155
useAprioriDirichlet() (*pyAgrum.BNLearner method*), 156
useAprioriSmoothing() (*pyAgrum.BNLearner method*), 156
useEM() (*pyAgrum.BNLearner method*), 156
useGreedyHillClimbing() (*pyAgrum.BNLearner method*), 156
useK2() (*pyAgrum.BNLearner method*), 156
useLocalSearchWithTabuList() (*pyAgrum.BNLearner method*), 156
useMDL() (*pyAgrum.BNLearner method*), 156
useMIIC() (*pyAgrum.BNLearner method*), 156
useNML() (*pyAgrum.BNLearner method*), 156
useNoApriori() (*pyAgrum.BNLearner method*), 156
useNoCorr() (*pyAgrum.BNLearner method*), 156
useScoreAIC() (*pyAgrum.BNLearner method*), 156
useScoreBD() (*pyAgrum.BNLearner method*), 156
useScoreBDeu() (*pyAgrum.BNLearner method*), 156
useScoreBIC() (*pyAgrum.BNLearner method*), 156
useScoreK2() (*pyAgrum.BNLearner method*), 156
useScoreLog2Likelihood() (*pyAgrum.BNLearner method*), 156
utility() (*pyAgrum.InfluenceDiagram method*), 176
utilityNodeSize() (*pyAgrum.InfluenceDiagram method*), 176

V

val() (*pyAgrum.Instantiation method*), 38
var_dims (*pyAgrum.Potential attribute*), 45
var_names (*pyAgrum.Potential attribute*), 45
variable() (*pyAgrum.BayesNet method*), 60
variable() (*pyAgrum.BayesNetFragment method*), 74
variable() (*pyAgrum.InfluenceDiagram method*), 176
variable() (*pyAgrum.Instantiation method*), 38
variable() (*pyAgrum.MarkovNet method*), 162
variable() (*pyAgrum.Potential method*), 45
VariableElimination (*class in pyAgrum*), 88
variableFromName() (*pyAgrum.BayesNet method*), 60
variableFromName() (*pyAgrum.BayesNetFragment method*), 74
variableFromName() (*pyAgrum.InfluenceDiagram method*), 176
variableFromName() (*pyAgrum.MarkovNet method*), 162
variableNodeMap() (*pyAgrum.BayesNet method*), 60
variableNodeMap() (*pyAgrum.BayesNetFragment method*), 74
variableNodeMap() (*pyAgrum.InfluenceDiagram method*), 176
variableNodeMap() (*pyAgrum.MarkovNet method*), 162

variablesSequence() (*pyAgrum.Instantiation method*), 38
variablesSequence() (*pyAgrum.Potential method*), 45
varNames (*pyAgrum.causal.ASTjointProba attribute*), 206
varOrder() (*pyAgrum.BNDatabaseGenerator method*), 61
varOrderNames() (*pyAgrum.BNDatabaseGenerator method*), 61
vars (*pyAgrum.causal.ASTposteriorProba attribute*), 206
varType() (*pyAgrum.DiscreteVariable method*), 23
varType() (*pyAgrum.DiscretizedVariable method*), 28
varType() (*pyAgrum.LabelizedVariable method*), 25
varType() (*pyAgrum.RangeVariable method*), 31
verbosity() (*pyAgrum.BNLearner method*), 156
verbosity() (*pyAgrum.CNLoopyPropagation method*), 197
verbosity() (*pyAgrum.CNMonteCarloSampling method*), 194
verbosity() (*pyAgrum.GibbsBNdistance method*), 65
verbosity() (*pyAgrum.GibbsSampling method*), 106
verbosity() (*pyAgrum.ImportanceSampling method*), 125
verbosity() (*pyAgrum.LoopyBeliefPropagation method*), 100
verbosity() (*pyAgrum.LoopyGibbsSampling method*), 131
verbosity() (*pyAgrum.LoopyImportanceSampling method*), 150
verbosity() (*pyAgrum.LoopyMonteCarloSampling method*), 137
verbosity() (*pyAgrum.LoopyWeightedSampling method*), 143
verbosity() (*pyAgrum.MonteCarloSampling method*), 112
verbosity() (*pyAgrum.WeightedSampling method*), 119
VI() (*pyAgrum.LazyPropagation method*), 75
VI() (*pyAgrum.ShaferShenoyInference method*), 82
VI() (*pyAgrum.ShaferShenoyMNInference method*), 163

W

WeightedSampling (*class in pyAgrum*), 113
what() (*pyAgrum.DefaultInLabel method*), 239
what() (*pyAgrum.DuplicateElement method*), 240
what() (*pyAgrum.DuplicateLabel method*), 240
what() (*pyAgrum.EmptyBSTree method*), 240
what() (*pyAgrum.EmptySet method*), 241
what() (*pyAgrum.FatalError method*), 241
what() (*pyAgrum.FormatNotFound method*), 242
what() (*pyAgrum.GraphError method*), 242
what() (*pyAgrum.GumException method*), 241
what() (*pyAgrum.IdError method*), 243
what() (*pyAgrum.InvalidArc method*), 243
what() (*pyAgrum.InvalidArgument method*), 243
what() (*pyAgrum.InvalidArgumentsNumber method*), 244
what() (*pyAgrum.InvalidDirectedCycle method*), 244
what() (*pyAgrum.InvalidEdge method*), 244
what() (*pyAgrum.InvalidNode method*), 245
what() (*pyAgrum.IOError method*), 242
what() (*pyAgrum.NoChild method*), 245
what() (*pyAgrum.NoNeighbour method*), 245
what() (*pyAgrum.NoParent method*), 246
what() (*pyAgrum.NotFound method*), 246
what() (*pyAgrum.NullElement method*), 246
what() (*pyAgrum.OperationNotAllowed method*), 247
what() (*pyAgrum.OutOfBounds method*), 247
what() (*pyAgrum.OutOfLowerBound method*), 247
what() (*pyAgrum.OutOfUpperBound method*), 248
what() (*pyAgrum.ReferenceError method*), 248
what() (*pyAgrum.SizeError method*), 248
what() (*pyAgrum.SyntaxError method*), 249
what() (*pyAgrum.UndefinedElement method*), 249
what() (*pyAgrum.UndefinedIteratorKey method*), 250
what() (*pyAgrum.UndefinedIteratorValue method*), 250
what() (*pyAgrum.UnknownLabelInDatabase method*), 250
whenArcAdded() (*pyAgrum.BayesNetFragment method*), 74
whenArcDeleted() (*pyAgrum.BayesNetFragment method*), 74
whenNodeAdded() (*pyAgrum.BayesNetFragment method*), 74
whenNodeDeleted() (*pyAgrum.BayesNetFragment method*), 74
with_traceback() (*pyAgrum.causal.HedgeException method*), 207
with_traceback() (*pyAgrum.causal.UnidentifiableException method*), 207
with_traceback() (*pyAgrum.DefaultInLabel method*), 239
with_traceback() (*pyAgrum.DuplicateElement method*), 240
with_traceback() (*pyAgrum.DuplicateLabel method*), 240
with_traceback() (*pyAgrum.EmptyBSTree method*), 240
with_traceback() (*pyAgrum.EmptySet method*), 241
with_traceback() (*pyAgrum.FatalError method*), 241
with_traceback() (*pyAgrum.FormatNotFound*

method), 242
with_traceback() (pyAgrum.GraphError method), 242
with_traceback() (pyAgrum.GumException method), 241
with_traceback() (pyAgrum.IdError method), 243
with_traceback() (pyAgrum.InvalidArc method), 243
with_traceback() (pyAgrum.InvalidArgument method), 243
with_traceback() (pyAgrum.InvalidArgumentsNumber method), 244
with_traceback() (pyAgrum.InvalidDirectedCycle method), 244
with_traceback() (pyAgrum.InvalidEdge method), 244
with_traceback() (pyAgrum.InvalidNode method), 245
with_traceback() (pyAgrum.IOError method), 242
with_traceback() (pyAgrum.NoChild method), 245
with_traceback() (pyAgrum.NoNeighbour method), 245
with_traceback() (pyAgrum.NoParent method), 246
with_traceback() (pyAgrum.NotFound method), 246
with_traceback() (pyAgrum.NullElement method), 246
with_traceback() (pyAgrum.OperationNotAllowed method), 247
with_traceback() (pyAgrum.OutOfBounds method), 247
with_traceback() (pyAgrum.OutOfLowerBound method), 247
with_traceback() (pyAgrum.OutOfUpperBound method), 248
with_traceback() (pyAgrum.ReferenceError method), 248
with_traceback() (pyAgrum.SizeError method), 248
with_traceback() (pyAgrum.SyntaxError method), 249
with_traceback() (pyAgrum.UndefinedElement method), 249
with_traceback() (pyAgrum.UndefinedIteratorKey method), 250
with_traceback() (pyAgrum.UndefinedIteratorValue method), 250
with_traceback() (pyAgrum.UnknownLabelInDatabase method), 250

X

XYfromCSV() (pyAgrum.skbn.BNClassifier method), 211