

# **pyAgrum Documentation**

***Release 0.16.4***

**Pierre-Henri Wuillemin (Sphinx)**

**December 15, 2019**



<b>1</b>	<b>Bayesian Network</b>	<b>3</b>
1.1	Model . . . . .	3
1.2	Tools for Bayesian networks . . . . .	16
1.3	Inference . . . . .	30
1.4	Exact Inference . . . . .	30
1.5	Approximated Inference . . . . .	50
1.6	Learning . . . . .	105
<b>2</b>	<b>Graphs manipulation</b>	<b>113</b>
2.1	Edges and Arcs . . . . .	113
2.2	Directed Graphs . . . . .	114
2.3	Undirected Graphs . . . . .	119
2.4	Mixed Graph . . . . .	126
<b>3</b>	<b>Random Variables</b>	<b>131</b>
3.1	Common API for Random Discrete Variables . . . . .	131
3.2	Concrete classes for Random Discrete Variables . . . . .	133
<b>4</b>	<b>Potential and Instantiation</b>	<b>143</b>
4.1	Instantiation . . . . .	144
4.2	Potential . . . . .	149
<b>5</b>	<b>Module notebook</b>	<b>157</b>
5.1	Helpers . . . . .	163
5.2	Visualization of Potentials . . . . .	163
5.3	Visualization of graphs . . . . .	164
5.4	Visualization of graphical models . . . . .	164
5.5	Visualization of approximation algorithm . . . . .	167
<b>6</b>	<b>Module bn2graph</b>	<b>169</b>
6.1	Visualization of Potentials . . . . .	171
6.2	Visualization of Bayesian Networks . . . . .	171
6.3	Hi-level functions . . . . .	172
<b>7</b>	<b>Module dynamic bayesian network</b>	<b>173</b>
<b>8</b>	<b>other pyAgrum.lib modules</b>	<b>175</b>
8.1	bn2roc . . . . .	175
8.2	bn2csv . . . . .	175
8.3	bn2scores . . . . .	177
8.4	bn_vs_bn . . . . .	177
8.5	pretty_print . . . . .	178

<b>9</b>	<b>pyAgrum.causal documentation</b>	<b>179</b>
9.1	Causal Model . . . . .	179
9.2	Causal Formula . . . . .	180
9.3	Causal Inference . . . . .	181
9.4	Abstract Syntax Tree for Do-Calculus . . . . .	182
9.5	Exceptions . . . . .	186
9.6	Notebook's tools for causality . . . . .	186
<b>10</b>	<b>Probabilistic Relational Models</b>	<b>189</b>
<b>11</b>	<b>Credal Networks</b>	<b>195</b>
11.1	Model . . . . .	195
11.2	Inference . . . . .	200
<b>12</b>	<b>Influence Diagram</b>	<b>207</b>
12.1	Model . . . . .	207
12.2	Inference . . . . .	213
<b>13</b>	<b>Functions from pyAgrum</b>	<b>215</b>
13.1	Input/Output for bayesian networks . . . . .	215
13.2	Input for influence diagram . . . . .	216
<b>14</b>	<b>Other functions from aGrUM</b>	<b>217</b>
14.1	Listeners . . . . .	217
14.2	Random functions . . . . .	218
14.3	OMP functions . . . . .	219
<b>15</b>	<b>Exceptions from aGrUM</b>	<b>221</b>
<b>16</b>	<b>Indices and tables</b>	<b>233</b>
	<b>Python Module Index</b>	<b>235</b>
	<b>Index</b>	<b>237</b>

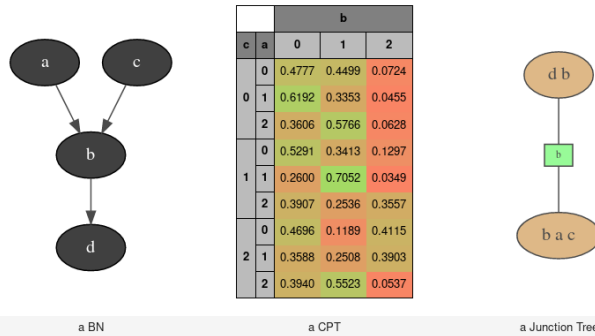
pyAgrum is a Python wrapper for the C++ aGrUM (<http://agrum.org>) library. It provides a high-level interface to the C++ part of aGrUM allowing to create, manage and perform efficient computations with Bayesian Networks.



(<http://agrum.org>)

```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

bn=gum.fastBN("a->b<-c;b->d",3)
gnb.sideBySide(bn,
               bn.cpt("b"),
               gnb.getJunctionTree(bn),
               captions=['a BN','a CPT','a Junction Tree'])
```



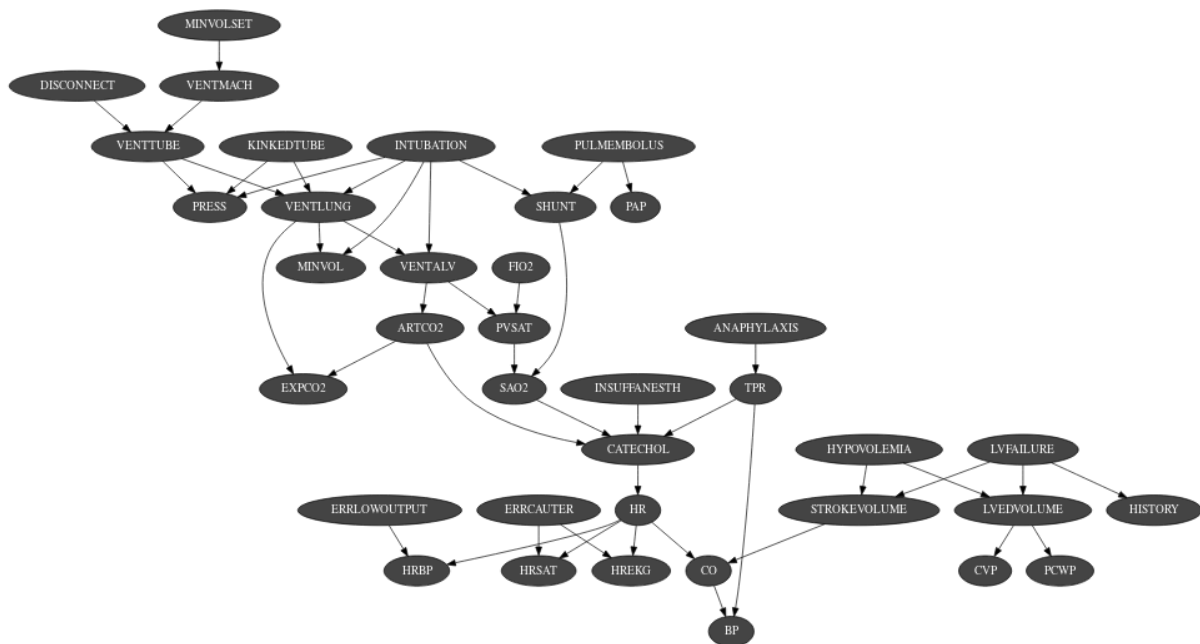
The module is generated using the SWIG (<https://www.swig.org>) interface generator. Custom-written code was added to make the interface more user friendly.

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) (<https://pyagrum.readthedocs.io>),
- [tutorials as jupyter notebooks](http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/01-tutorial.ipynb.html) (<http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/01-tutorial.ipynb.html>),
- a [gitlab repository](https://gitlab.com/agrumery/aGrUM) (<https://gitlab.com/agrumery/aGrUM>),
- and a [website](http://agrum.org) (<http://agrum.org>).





The Bayesian Network is the main object of pyAgrum. A Bayesian network is a probabilistic graphical model. 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 independencies in the joint distribution. These conditional independencies allow to factorize the joint distribution, thereby allowing to compactly represent very large ones. Moreover, inference algorithms can also use this graph to speed up the computations. Finally, the Bayesian networks can be learnt from data.

## 1.1 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 131)) – 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 131)) – the variable to be added

**Returns** the id of the added value

**Return type** int

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

Add a variable, its associate node and an AND implementation.

The id of the new variable is automatically generated.

**Parameters** **variable** (*pyAgrum.DiscreteVariable* (page 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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 131)) – The variable added by copy

**Returns** the id of the added variable.

**Return type** `int`

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

**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.

**arcs** (*BayesNet self*)

**Returns** The list 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 149)) – 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

**completeInstantiation** (*DAGmodel self*)

Get an instantiation over all the variables of the model.

**Returns** the complete instantiation

**Return type** pyAgrum.instantiation

**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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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 149)

**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 117)

**dim** (*IBayesNet self*)

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

**Returns** the dimension of the BayesNet

**Return type** int

**empty** (*DAGmodel self*)

**Returns** True if the model is empty

**Return type** bool

**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 3)

**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 131)) – 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 113)) – 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)

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

Create a bn with a dotlike syntax : 'a->b->c;b->d;'.

The domain size maybe specified using 'a[10]'.

Note that if the dotlike string contains such a specification for an already defined variable, the first specification will be used.

#### 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 3)

**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** ()

Deprecated method in pyAgrum>0.12.0. See nodes instead.

**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 LabeledVariable.

**Parameters**

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

**Raises**

- `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*)

**Returns** The log10 domain size of the joint probability for the model.

**Return type** double

**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 119)

**names** (*BayesNet self*)

**Returns** The names of the graph variables

**Return type** list

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

**Parameters** **var** (`pyAgrum.DiscreteVariable` (page 131)) – 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

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

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

**Returns** the set of the parents ids.

**Return type** Set

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

**Warning:** Unreferenced function

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

**Warning:** Unreferenced function

**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 113)) – an arc

**Raises** `gum.InvalidArc` – If the arc does not exist 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 LabeledVariable.

**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** (*DAGmodel self, str name, str value*)

**Warning:** Unreferenced function

**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 131)

**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 131)

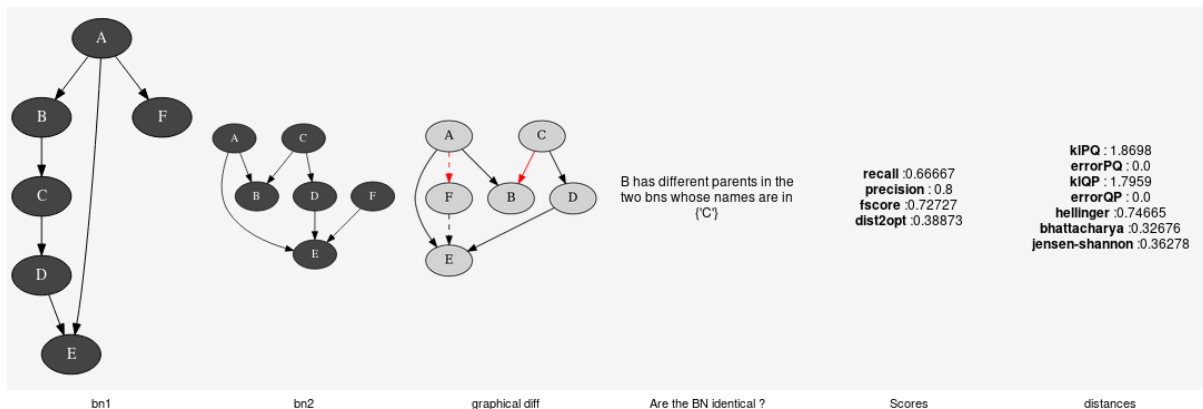
**Raises** `gum.IndexError` – If the graph does not contain the variable

**variableNodeMap** (*BayesNet self*)

**Returns** the variable node map

**Return type** *pyAgrum.variableNodeMap*

## 1.2 Tools for Bayesian networks



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

### 1.2.1 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=","*)

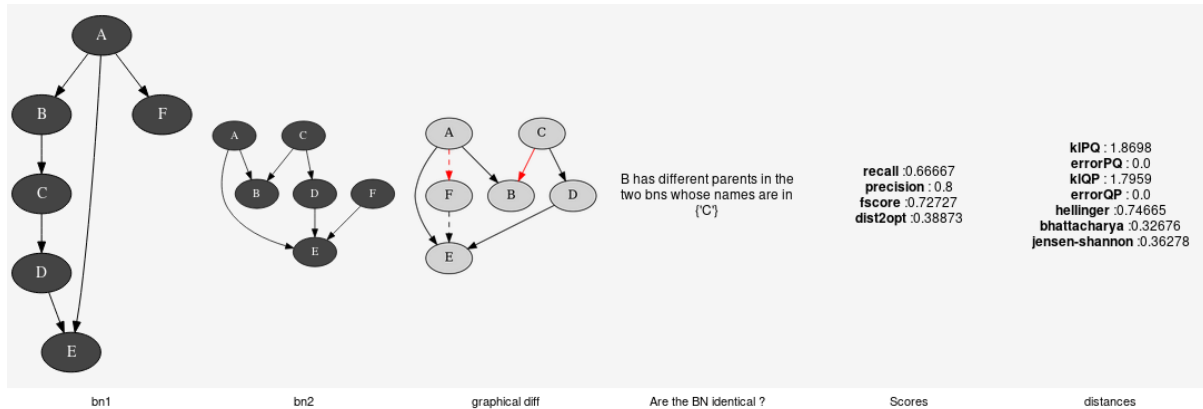
**toCSV** (*BNDatabaseGenerator self*, *str csvFileURL*, *bool useLabels=True*, *bool append=False*, *str csvSeparator=","*, *bool checkOnAppend=False*)

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

**varOrder** (*BNDatabaseGenerator self*)

**varOrderNames** (*BNDatabaseGenerator self*)

## 1.2.2 Comparison of Bayesian networks



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

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

Class representing exact computation of divergence and distance between BNs

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

**Parameters:**

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

**ExactBNdistance(ebnd) -> ExactBNdistance**

**Parameters:**

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

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

**compute** (*ExactBNdistance self*)

**Returns** a dictionary 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 dictionary 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*)

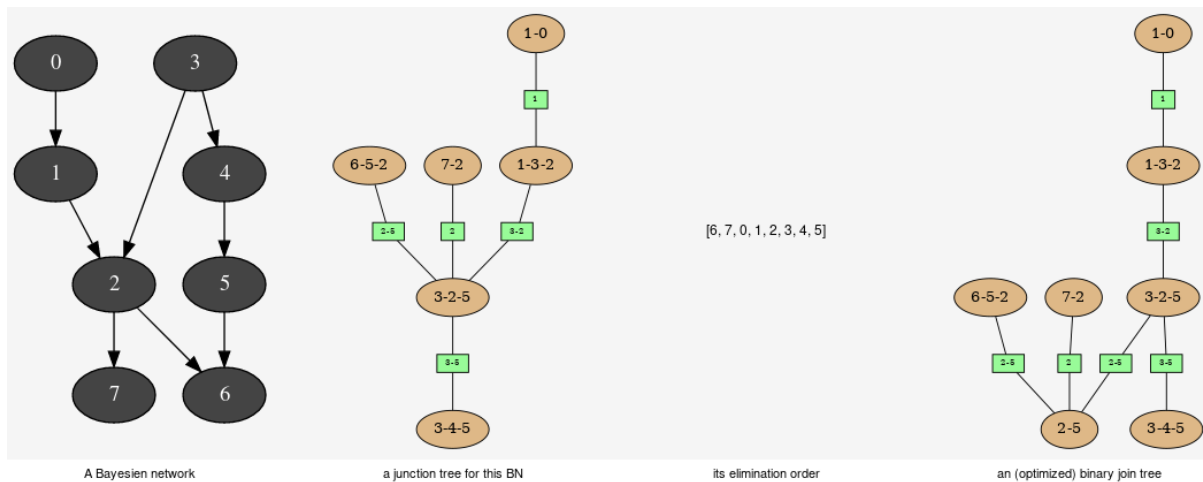
Returns True if the verbosity is enabled

Return type `bool`

### 1.2.3 Explanation and analysis

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





**class** `pyAgrum.JunctionTreeGenerator`

JunctionTreeGenerator is use to generate junction tree or binary junction tree from bayesian networks.

**JunctionTreeGenerator()** -> **JunctionTreeGenerator** default constructor

**binaryJoinTree** (*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 heuristics assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

#### Parameters

- **g** (`pyAgrum.UndiGraph` (page 119)) – a undirected graph
- **dag** (`pyAgrum.DAG` (page 117)) – a dag
- **bn** (`pyAgrum.BayesNet` (page 3)) – a BayesianNetwork
- **partial\_order** (`List[List[int]]`) – a partial order among the nodeIDs

**Returns** the current binary joint tree

**Return type** `pyAgrum.CliqueGraph` (page 122)

**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 heuristics assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

#### Parameters

- **g** (`pyAgrum.UndiGraph` (page 119)) – a undirected graph
- **dag** (`pyAgrum.DAG` (page 117)) – a dag
- **bn** (`pyAgrum.BayesNet` (page 3)) – a BayesianNetwork
- **partial\_order** (`List[List[int]]`) – a partial order among the nodeIDs

**Returns** the current elimination order.

**Return type** `pyAgrum.CliqueGraph` (page 122)

```
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) -> Clique-  
Graph
```

Computes the junction tree for its parameters. If the first parameter is a graph, the heuristic assumes that all the nodes 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 119)) – a undirected graph
- **dag** ([pyAgrum.DAG](#) (page 117)) – a dag
- **bn** ([pyAgrum.BayesNet](#) (page 3)) – a BayesianNetwork
- **partial\_order** (*List[List[int]]*) – a partial order among the nodeIDs

**Returns** the current junction tree.

**Return type** [pyAgrum.CliqueGraph](#) (page 122)

```
class pyAgrum.EssentialGraph (*args)  
Proxy of C++ pyAgrum.EssentialGraph class.
```

```
arcs (EssentialGraph self)
```

**Returns** The list 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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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

```
ids ()
```

Deprecated method in pyAgrum>0.12.0. See nodes instead.

```
mixedGraph (EssentialGraph self)
```

**Returns** the mixed graph

**Return type** [pyAgrum.MixedGraph](#) (page 126)

**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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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 117)

**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

## 1.2.4 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 referred 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

**arcs** (*BayesNetFragment self*)

**Returns** The list 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 consistent for this variable. This function checks the consistency for a variable or for all.

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

**Returns** True if the variable(s) is consistent.

**Return type** boolean

**Raises** `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** (*DAGmodel self*)

Get an instantiation over all the variables of the model.

**Returns** the complete instantiation

**Return type** `pyAgrum.instantiation`

**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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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 149)

**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 117)

**dim** (*IBayesNet self*)

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

**Returns** the dimension of the BayesNet

**Return type** int

**empty** (*DAGmodel self*)

**Returns** True if the model is empty

**Return type** bool

**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

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

installAscendants(BayesNetFragment self, str name)

Add the variable and all its ascendants in the fragment. No inconsistent node are created.

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

**Raises** 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 149)) – 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 149)) – 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.

**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** (*DAGmodel self*)

**Returns** The log10 domain size of the joint probability for the model.

**Return type** double

**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 119)

**names** (*BayesNetFragment self*)

**Returns** The names of the graph variables

**Return type** list

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

**Parameters** **var** ([pyAgrum.DiscreteVariable](#) (page 131)) – 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

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

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

**Returns** the set of the parents ids.

**Return type** Set



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

**Warning:** Unreferenced function

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

**Warning:** Unreferenced function

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

**Warning:** Unreferenced function

**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 inconsistent.

**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 inconsistent.

**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 131)**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 131)**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*)

## 1.3 Inference

Inference is the process that consists in computing new probabilistic information from a Bayesian network and some evidence. aGrUM/pyAgrum mainly focus and the computation of (joint) posterior for some variables of the Bayesian networks given soft or hard evidence that are the form of likelihoods on some variables. Inference is a hard task (NP-complete). aGrUM/pyAgrum implements exact inference but also approximated inference that can converge slowly and (even) not exactly but that can in many cases be useful for applications.

## 1.4 Exact Inference

### 1.4.1 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*)

**Parameters**

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

**Returns** the computed Shanon’s entropy of a node given the observation

**Return type** double

**VI** (*LazyPropagation self, int X, int Y*)

**Parameters**

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

**Returns** variation of information between X and Y

**Return type** double

**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, int target, PyObject \* evs*)

`evidenceImpact(LazyPropagation self, str target, Vector_string evs) -> Potential`

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**evidenceJointImpact** (*LazyPropagation self, PyObject \* targets, PyObject \* evs*)

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

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

**Parameters**

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

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

**Return type** `pyAgrum.Potential` (page 149)

**Raises** `gum.Exception` – If some evidence 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 122)

**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 by the `Potential`.

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

**Return type** *pyAgrum.Potential* (page 149)

**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 122)

**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`

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 149)

**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

## 1.4.2 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*)

**Parameters**

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

**Returns** the computed Shanon's entropy of a node given the observation

**Return type** double

**VI** (*ShaferShenoyInference self, int X, int Y*)

**Parameters**

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

**Returns** variation of information between X and Y

**Return type** double

**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, int target, PyObject \* evs*)  
`evidenceImpact(ShaferShenoyInference self, str target, Vector_string evs) -> Potential`

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

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

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

**Parameters**

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

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

**Return type** `pyAgrum.Potential` (page 149)

**Raises** `gum.Exception` – If some evidence 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 122)

**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 by the Potential.

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

**Return type** *pyAgrum.Potential* (page 149)

**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 122)

**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

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 149)

**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

### 1.4.3 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 Shannon'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, int target, PyObject \* evs*)

`evidenceImpact(VariableElimination self, str target, Vector_string evs) -> Potential`

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

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

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

**Parameters**

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

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

**Return type** `pyAgrum.Potential` (page 149)

**Raises** `gum.Exception` – If some evidence 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 by the `Potential`.

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

**Return type** *pyAgrum.Potential* (page 149)

**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 122)

**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 149)

**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

## 1.5 Approximated Inference

### 1.5.1 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 Shannon'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*) **chgEvi-**  
**dence**(*LoopyBeliefPropagation self, int id, str val*) **chgEvidence**(*LoopyBeliefPropagation self,*  
*str nodeName, str val*) **chgEvidence**(*LoopyBeliefPropagation self, int id, Vector vals*) **chgEvi-**  
**dence**(*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, int target, PyObject \* evs*)

evidenceImpact(*LoopyBeliefPropagation self, str target, Vector\_string evs*) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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 149)

**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

## 1.5.2 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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)

evidenceImpact(*GibbsSampling self, str target, Vector\_string evs*) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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 149)

**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  $max \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $timeout \leq 0.0$

**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 size 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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)  
 evidenceImpact(MonteCarloSampling self, str target, Vector\_string evs) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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 149)

**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  $\text{eps} < 0$

**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  $\text{max} \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $\text{timeout} \leq 0.0$

**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 size 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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)

evidenceImpact(*WeightedSampling self, str target, Vector\_string evs*) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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 149)

**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  $\text{eps} < 0$

**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  $\text{max} \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $\text{timeout} \leq 0.0$

**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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)  
 evidenceImpact(ImportanceSampling self, str target, Vector\_string evs) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** *pyAgrum.Potential* (page 149)

**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 149)

**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  $\text{eps} < 0$

**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`

### 1.5.3 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 size 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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)  
 evidenceImpact(*LoopyGibbsSampling self, str target, Vector\_string evs*) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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.

**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 149)

**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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)

evidenceImpact(*LoopyMonteCarloSampling self, str target, Vector\_string evs*) -> Potential

Create a *pyAgrum.Potential* for P(target|evs) (for all instantiation of target and evs)

#### Parameters

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

**Returns** a Potential for  $P(\text{targets}|\text{eivs})$

**Return type** *pyAgrum.Potential* (page 149)

**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.

**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 149)

**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  $max \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $timeout \leq 0.0$

**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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)  
evidenceImpact(*LoopyWeightedSampling self, str target, Vector\_string evs*) -> Potential

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** *pyAgrum.Potential* (page 149)

**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.

**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 149)

**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  $\text{eps} < 0$

**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  $\text{max} \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $\text{timeout} \leq 0.0$

**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 size 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 149)

**Raises** *UndefinedElement* (page 231) – 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, int target, PyObject \* evs*)

`evidenceImpact(LoopyImportanceSampling self, str target, Vector_string evs) -> Potential`

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

**Parameters**

- **target** (*set*) – a set of targets ids or names.
- **evs** (*set*) – a set of nodes ids or names.

**Warning:** if some evs are d-separated, they are not included in the Potential.

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

**Return type** `pyAgrum.Potential` (page 149)

**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.

**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 149)

**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  $\text{eps} < 0$

**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  $\text{max} \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $\text{timeout} \leq 0.0$

**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`

## 1.6 Learning

pyAgrum encloses all the learning processes for Bayesian network in a simple class `BNLearner`. This class gives access directly to the complete learning algorithm and theirs parameters (such as prior, scores, constraints, etc.) but also proposes low-level functions that eases the work on developping new learning algorithms (for instance, compute `chi2` or conditionnl likelihood on the database, etc.).

**class** `pyAgrum.BNLearner` (*filename*)

**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 113)) – 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 113)) – 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*)

**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 3)

**learnDAG** (*BNLearner self*)

learn a structure from a file

**Returns** the learned DAG

**Return type** [\*pyAgrum.DAG\*](#) (page 117)

**learnMixedStructure** (*BNLearner self*)

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

**Returns** the learned structure as an EssentialGraph

**Return type** [\*pyAgrum.EssentialGraph\*](#) (page 22)

**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 117)) –
- **bn** ([pyAgrum.BayesNet](#) (page 3)) –
- **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 3)

**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 number 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  $p < 1$

**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  $\text{eps} < 0$

**setInitialDAG** (*BNLearner self, DAG g*)

**Parameters** **dag** (`pyAgrum.DAG` (page 117)) – 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  $\text{max} \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $\text{timeout} \leq 0.0$

**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 less 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

---

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

## 2.1 Edges and Arcs

### 2.1.1 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`

## 2.1.2 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** (*pyAgrum.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`

## 2.2 Directed Graphs

### 2.2.1 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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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



**ids** ()  
 Deprecated method in pyAgrum>0.12.0. See nodes instead.

**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

## 2.2.2 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

**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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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** (*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

**ids** ()

Deprecated method in pyAgrum>0.12.0. See nodes instead.

**moralGraph** (*DAG self*)

**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

## 2.3 Undirected Graphs

### 2.3.1 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 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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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

**ids** ()

Deprecated method in pyAgrum>0.12.0. See nodes instead.

**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

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

**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 119)

**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

## 2.3.2 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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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

**ids** ()

Deprecated method in pyAgrum>0.12.0. See nodes instead.

**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

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

**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 119)

**separator** (*CliqueGraph self, int cliq1, int cliq2*)

**Parameters**

- **edge** (*pyAgrum.Edge* (page 114)) – the edge to be checked
- **cliq1** (*int*) – one extremity of the edge
- **cliq2** (*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 3)) –
- **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

## 2.4 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.

**Parameters** **graph** (*pyAgrum's graph*) – A graph, a Bayesian network, more generally an object that has *nodes*, *children/parents* or *neighbours* methods in which the search will take place

**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 between n1 and n2.

**Parameters**

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

**Returns** True if the arc exists

**Return type** bool

**existsEdge** (*MixedGraph self, int n1, int n2*)

Check if an edge exists between n1 and n2.

**Parameters**

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

**Returns** True if the arc exists

**Return type** bool

**existsNode** (*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 directed path exists between from and to.

**Parameters**

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

**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

**ids** ()

Deprecated method in pyAgrim>0.12.0. See nodes instead.

**mixedOrientedPath** (*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 (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

**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 nodesSet*)

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

**Returns** The partial graph formed by the nodes given in parameter

**Return type** [pyAgrim.UndiGraph](#) (page 119)

**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

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.

### 3.1 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 136)

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

**toLabelizedVar** (*DiscreteVariable self*)

**Returns** the labeled variable

**Return type** [\*pyAgrum.LabelizedVariable\*](#) (page 133)

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

**toRangeVar** (*DiscreteVariable self*)

**Returns** the range variable

**Return type** [\*pyAgrum.RangeVariable\*](#) (page 138)

**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

## 3.2 Concrete classes for Random Discrete Variables

### 3.2.1 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 labeled 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 labeled variable**Returns** the labeled variable**Return type** *pyAgrum.LabelizedVariable* (page 133)**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 labeled 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 136)

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

**toLabelizedVar** (*DiscreteVariable self*)

**Returns** the labeled variable

**Return type** [pyAgrum.LabelizedVariable](#) (page 133)

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

**toRangeVar** (*DiscreteVariable self*)

**Returns** the range variable

**Return type** [pyAgrum.RangeVariable](#) (page 138)

**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: LabeledVariable, 2: RangeVariable

**Return type** int

### 3.2.2 DiscretizedVariable

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

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

**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 136)

**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 136)

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

**toLabelizedVar** (*DiscreteVariable self*)

**Returns** the labeled variable

**Return type** *pyAgrum.LabelizedVariable* (page 133)

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

**toRangeVar** (*DiscreteVariable self*)

**Returns** the range variable

**Return type** *pyAgrum.RangeVariable* (page 138)

**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`

### 3.2.3 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 136)



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

**toLabeledVar** (*DiscreteVariable self*)

**Returns** the labeled variable

**Return type** `pyAgrum.LabeledVariable` (page 133)

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

**toRangeVar** (*DiscreteVariable self*)

**Returns** the range variable

**Return type** `pyAgrum.RangeVariable` (page 138)

**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: `LabeledVariable`, 2: `RangeVariable`

**Return type** `int`



---

## Potential and Instantiation

---

*pyAgrum.Potential* (page 149) is a multi-dimensional array with a *pyAgrum.DiscreteVariable* (page 131) 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 149) with the sequence of *pyAgrum.DiscreteVariable* (page 131) 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 149) 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 149) include tensor operators (see for instance this [notebook](http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/05-potentials.ipynb.html) (<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

```

## 4.1 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 131)) – The new variable added to the Instantiation

**Raises** `DuplicateElement` (page 221) – 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 131) 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 144)

**Raises**

- *NotFound* (page 228) – If variable *v* does not belong to the instantiation.
- *OutOfBounds* – 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, DiscreteVariable v) -> bool

Indicates whether a given variable belongs to the Instantiation.

**Parameters** *v* (*pyAgrum.DiscreteVariable* (page 131)) – 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 144)) – 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 131)) – 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 144)) – 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 131)) – The variable to decrement in this Instantiation.

**Raises** *NotFound* (page 228) – 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*)

**Parameters** **v** ([pyAgrum.DiscreteVariable](#) (page 131)) – The variable to be removed from this Instantiation.

**Raises** [NotFound](#) (page 228) – 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 144)) – 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 131)) – The variable not to increment in this Instantiation.

**incOut** (*Instantiation self, Instantiation i*)

Operator ++ for the variables not in i.

**Parameters** **i** ([Instantiation](#) (page 144)) – 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 131)) – The variable to increment in this Instantiation.

**Raises** [NotFound](#) (page 228) – If variable v does not belong to the Instantiation.

**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 131)) – the variable for which its position is return.

**Raises** [NotFound](#) (page 228) – 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 144)) – 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 144)) – 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 131)) – 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 144)) – 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 131)) – 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 144)) – 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 131)) – 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 144)) – 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 131)) – The variable that will be set to its last value in this Instantiation.

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

**Parameters** **i** (`pyAgrum.Instantiation` (page 144)) – An Instantiation in which the new values are searched

**Returns** a reference to the instantiation

**Return type** `pyAgrum.Instantiation` (page 144)

**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

**Parameters**

- **i** (*int*) – The index of the variable.
- **var** (`pyAgrum.DiscreteVariable` (page 131)) – The variable the value of which we wish to know

**Returns** the current value of the variable.

**Return type** int

**Raises** `NotFound` (page 228) – 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 131)

**Raises** `NotFound` (page 228) – If the element cannot be found.

**variablesSequence** (*Instantiation self*)

**Returns** the sequence of DiscreteVariable of this instantiation.

**Return type** List



## 4.2 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 149)) – 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 149)

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

Add a discrete variable to the potential.

**Parameters** **v** (`pyAgrum.DiscreteVariable` (page 131)) – the var to be added

**Raises**

- `DuplicateElement` (page 221) – If the variable is already in this Potential.
- `InvalidArgument` (page 225) – If the variable is empty.

**argmax** (*Potential self*)

**argmin** (*Potential self*)

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

**Parameters** **v** (`pyAgrum.Potential` (page 149)) – a DiscreteVariable.

**Returns** True if the var is in the potential

**Return type** bool

**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 dictionary containing discrete variables (?)

**Returns** the new Potential

**Return type** *pyAgrum.Potential* (page 149)

**fill** (*v*)

Deprecated method in pyAgrum>0.12.0. See fillWith instead.

**fillWith** (*Potential self, Potential src*)

fillWith(Potential self, Potential src, Vector\_string mapSrc) -> 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 149)

**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 149)

**Raises** `gum.InvalidArgument` – If the first variable is Labeledized 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 144)) – an Instantiation

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

**Return type** double

**isNonZeroMap** (*Potential self*)

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

**Return type** *pyAgrum.Potential* (page 149)

**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 149)

**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 149)

**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 149)

**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 149)

**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 149)

**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 149)

**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 149)

**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 149)

**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*)

**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 149)

**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 149)

**normalizeAsCPT** (*Potential self*)

Normalize the Potential as a CPT

**Returns** a reference to the normalized Potential

**Return type** [pyAgrum.Potential](#) (page 149)

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

**populate** (*v*)

Deprecated method in pyAgrum>0.12.0. See `fillWith` instead.

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

**Parameters** **v** ([pyAgrum.DiscreteVariable](#) (page 131)) – 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, PyObject \* varname*)

**Parameters** **v** ([pyAgrum.DiscreteVariable](#) (page 131)) – The variable for which the index should be 0.

**Returns** a reference to the modified potential

**Return type** [pyAgrum.Potential](#) (page 149)

**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 131)) – The variable to be removed

**Returns** a reference to the modified potential

**Return type** [pyAgrum.Potential](#) (page 149)

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

**reorganize** (*Potential self, vector< pyAgrum.DiscreteVariable \*, allocator< pyAgrum.DiscreteVariable \*> > vars*)  
reorganize(Potential self, PyObject \* varnames) -> 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 149)

**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 144)) – 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

<b>Warning:</b> 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 variable 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





tools for BN analysis in jupyter notebook

`pyAgrum.lib.notebook.animApproximationScheme (apsc, scale=<ufunc 'log10'>)`  
show an animated version of an approximation algorithm

### Parameters

- **apsc** – the approximation algorithm
- **scale** – a function to apply to the figure

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

`pyAgrum.lib.notebook.getBN (bn, size=None, nodeColor=None, arcWidth=None, arcColor=None, cmap=None, cmapArc=None)`  
get a HTML string for a Bayesian network

### Parameters

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

**Returns** the graph

`pyAgrum.lib.notebook.getBNDiff (bn1, bn2, size=None)`  
get a HTML string representation of a graphical diff between the arcs of `_bn1` (reference) with those of `_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`

#### Parameters

- **bn1** ([BayesNet](#) (page 3)) – referent model for the comparison
- **bn2** ([BayesNet](#) (page 3)) – bn compared to the referent model
- **size** – size of the rendered graph

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

#### Parameters

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

**Returns** the HTML representation of the graph

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

#### Parameters

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

**Returns** the HTML representation of the graph as a string

`pyAgrum.lib.notebook.getInference(bn, engine=None, evs=None, targets=None, size=None, nodeColor=None, arcWidth=None, arcColor=None, cmap=None, cmapArc=None, dag=None)`  
get a HTML string for an inference in a notebook

#### Parameters

- **bn** (`gum.BayesNet`) –
- **engine** (`gum.Inference`) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (`dictionary`) – map of evidence
- **targets** (`set`) – set of targets
- **size** (`string`) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.

:param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

`pyAgrum.lib.notebook.getInferenceEngine(ie, inferenceCaption)`  
display an inference as a BN+ lists of hard/soft evidence and list of targets

**Parameters**

- **ie** (*gum.InferenceEngine*) – inference engine
- **inferenceCaption** (*string*) – title for caption

`pyAgrum.lib.notebook.getInfluenceDiagram(diag, size=None)`

get a HTML string for an influence diagram as a graph

**Parameters**

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns** the HTML representation of the influence diagram

`pyAgrum.lib.notebook.getInformation(bn, evs=None, size=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>)`

get a HTML string for a bn annotated with results from inference : entropy and mutual informations

**Parameters**

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

**Returns** the HTML string

`pyAgrum.lib.notebook.getJunctionTree(bn, withNames=True, size=None)`

get a HTML string for a junction tree (more specifically a join tree)

**Parameters**

- **bn** – the bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

**Returns** the HTML representation of the graph

`pyAgrum.lib.notebook.getPosterior(bn, evs, target)`

shortcut for `getProba(gum.getPosterior(bn, evs, target))`

**Parameters**

- **bn** (*gum.BayesNet*) – the BayesNet
- **evs** (*dict(str->int)*) – map of evidence
- **target** (*str*) – name of target variable

**Returns** the matplotlib graph

`pyAgrum.lib.notebook.getPotential(pot, digits=None, withColors=None, varnames=None)`

return a HTML string of a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

**Parameters**

- **pot** (*gum.Potential*) – the potential to get
- **digits** (*int*) – number of digits to show
- **of strings varnames** (*list*) – the aliases for variables name in the table

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

**Returns** the HTML string

`pyAgrum.lib.notebook.getSideBySide(*args, **kwargs)`  
create an HTML table for args as string (using `string`, `_repr_html_()` or `str()`)

**Parameters**

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

**Returns** a string representing the table

`pyAgrum.lib.notebook.showBN(bn, size=None, nodeColor=None, arcWidth=None, arcColor=None, cmap=None, cmapArc=None)`  
show a Bayesian network

**Parameters**

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

**Returns** the graph

`pyAgrum.lib.notebook.showBNDiff(bn1, bn2, size=None)`  
show a graphical diff between the arcs of `_bn1` (reference) with those of `_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`

**Parameters**

- **bn1** ([BayesNet](#) (page 3)) – referent model for the comparison
- **bn2** ([BayesNet](#) (page 3)) – bn compared to the referent model
- **size** – size of the rendered graph

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

**Parameters**

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

**Returns** the representation of the graph

`pyAgrum.lib.notebook.showGraph(gr, size=None)`  
show a pydot graph in a notebook

**Parameters**

- **gr** – pydot graph
- **size** – size of the rendered graph

**Returns** the representation of the graph

```
pyAgrum.lib.notebook.showInference (bn, engine=None, evs=None, targets=None,  
                                     size=None, nodeColor=None, arcWidth=None,  
                                     arcColor=None, cmap=None, cmapArc=None,  
                                     dag=None)
```

show pydot graph for an inference in a notebook

#### Parameters

- **bn** (*gum.BayesNet*) –
- **engine** (*gum.Inference*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dictionary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.

:param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

```
pyAgrum.lib.notebook.showInfluenceDiagram (diag, size=None)
```

show an influence diagram as a graph

#### Parameters

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns** the representation of the influence diagram

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

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

#### Parameters

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

**Returns** the graph

```
pyAgrum.lib.notebook.showJunctionTree (bn, withNames=True, size=None)
```

Show a junction tree

#### Parameters

- **bn** – the bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

**Returns** the representation of the graph

`pyAgrum.lib.notebook.showPosterior(bn, evs, target)`  
 shortcut for `showProba(gum.getPosterior(bn, evs, target))`

### Parameters

- **bn** – the BayesNet
- **evs** – map of evidence
- **target** – name of target variable

`pyAgrum.lib.notebook.showPotential(pot, digits=None, withColors=None, varnames=None)`

show a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

### Parameters

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

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

**Returns** the display of the potential

`pyAgrum.lib.notebook.showProba(p, scale=1.0)`

Show a mono-dim Potential

**Parameters** **p** – the mono-dim Potential

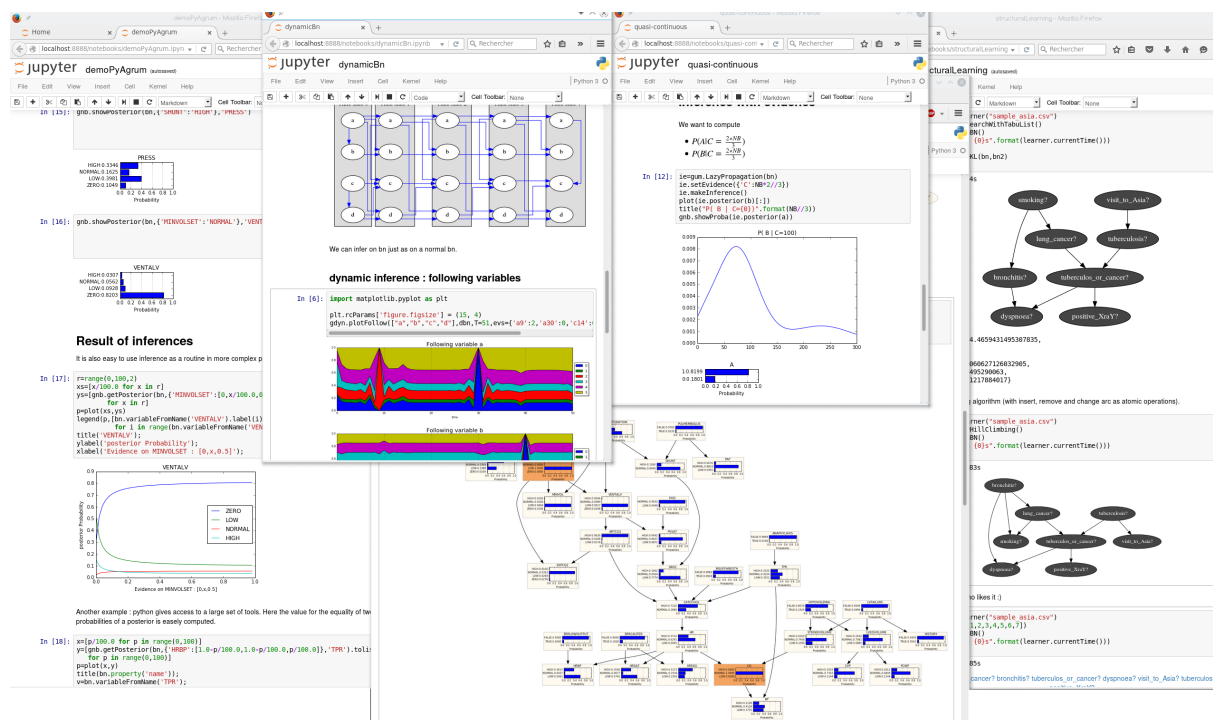
### Returns

`pyAgrum.lib.notebook.sideBySide(*args, **kwargs)`

display side by side args as HTML fragment (using string, `_repr_html_()` or `str()`)

### Parameters

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



## 5.1 Helpers

`pyAgrum.lib.notebook.configuration()`

Display the collection of dependance and versions

`pyAgrum.lib.notebook.sideBySide(*args, **kwargs)`

display side by side args as HTML fragment (using `string`, `_repr_html_()` or `str()`)

### Parameters

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

## 5.2 Visualization of Potentials

`pyAgrum.lib.notebook.showProba(p, scale=1.0)`

Show a mono-dim Potential

**Parameters** **p** – the mono-dim Potential

### Returns

`pyAgrum.lib.notebook.getPosterior(bn, evs, target)`

shortcut for `getProba(gum.getPosterior(bn, evs, target))`

### Parameters

- **bn** (`gum.BayesNet`) – the BayesNet
- **evs** (`dict(str->int)`) – map of evidence
- **target** (`str`) – name of target variable

**Returns** the matplotlib graph

`pyAgrum.lib.notebook.showPosterior(bn, evs, target)`

shortcut for `showProba(gum.getPosterior(bn, evs, target))`

### Parameters

- **bn** – the BayesNet
- **evs** – map of evidence
- **target** – name of target variable

`pyAgrum.lib.notebook.getPotential(pot, digits=None, withColors=None, varnames=None)`

return a HTML string of a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

### Parameters

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

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

**Returns** the HTML string

`pyAgrum.lib.notebook.showPotential(pot, digits=None, withColors=None, varnames=None)`

show a `gum.Potential` as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

### Parameters

- **pot** (*gum.Potential*) – the potential to get
- **digits** (*int*) – number of digits to show
- **of strings varnames** (*list*) – the aliases for variables name in the table

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

**Returns** the display of the potential

## 5.3 Visualization of graphs

`pyAgrum.lib.notebook.getDot(dotstring, size=None)`

get a dot string as a HTML string

### Parameters

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

**Returns** the HTML representation of the graph

`pyAgrum.lib.notebook.showDot(dotstring, size=None)`

show a dot string as a graph

### Parameters

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

**Returns** the representation of the graph

`pyAgrum.lib.notebook.getGraph(gr, size=None)`

get a HTML string representation of pydot graph

### Parameters

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

**Returns** the HTML representation of the graph as a string

`pyAgrum.lib.notebook.showGraph(gr, size=None)`

show a pydot graph in a notebook

### Parameters

- **gr** – pydot graph
- **size** – size of the rendered graph

**Returns** the representation of the graph

## 5.4 Visualization of graphical models

`pyAgrum.lib.notebook.getBN(bn, size=None, nodeColor=None, arcWidth=None, arcColor=None, cmap=None, cmapArc=None)`

get a HTML string for a Bayesian network



**Parameters**

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

**Returns** the graph

```
pyAgrum.lib.notebook.showBN(bn, size=None, nodeColor=None, arcWidth=None, arc-
                             Color=None, cmap=None, cmapArc=None)
```

show a Bayesian network

**Parameters**

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

**Returns** the graph

```
pyAgrum.lib.notebook.getInference(bn, engine=None, evs=None, targets=None, size=None,
                                   nodeColor=None, arcWidth=None, arcColor=None,
                                   cmap=None, cmapArc=None, dag=None)
```

get a HTML string for an inference in a notebook

**Parameters**

- **bn** (*gum.BayesNet*) –
- **engine** (*gum.Inference*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dictionary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.

:param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

```
pyAgrum.lib.notebook.showInference (bn, engine=None, evs=None, targets=None,
                                     size=None, nodeColor=None, arcWidth=None,
                                     arcColor=None, cmap=None, cmapArc=None,
                                     dag=None)
```

show pydot graph for an inference in a notebook

#### Parameters

- **bn** (*gum.BayesNet*) –
- **engine** (*gum.Inference*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dictionary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.

:param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

```
pyAgrum.lib.notebook.getJunctionTree (bn, withNames=True, size=None)
```

get a HTML string for a junction tree (more specifically a join tree)

#### Parameters

- **bn** – the bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

**Returns** the HTML representation of the graph

```
pyAgrum.lib.notebook.showJunctionTree (bn, withNames=True, size=None)
```

Show a junction tree

#### Parameters

- **bn** – the bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

**Returns** the representation of the graph

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

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

#### Parameters

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

**Returns** the graph

```
pyAgrum.lib.notebook.getInformation (bn, evs=None, size=None,  
                                     cmap=<matplotlib.colors.LinearSegmentedColormap  
                                     object>)
```

get a HTML string for a bn annotated with results from inference : entropy and mutual informations

**Parameters**

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

**Returns** the HTML string

```
pyAgrum.lib.notebook.showInfluenceDiagram (diag, size=None)
```

show an influence diagram as a graph

**Parameters**

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns** the representation of the influence diagram

```
pyAgrum.lib.notebook.getInfluenceDiagram (diag, size=None)
```

get a HTML string for an influence diagram as a graph

**Parameters**

- **diag** – the influence diagram
- **size** – size of the rendered graph

**Returns** the HTML representation of the influence diagram

## 5.5 Visualization of approximation algorithm

```
pyAgrum.lib.notebook.animApproximationScheme (apsc, scale=<ufunc 'log10'>)
```

show an animated version of an approximation algorithm

**Parameters**

- **apsc** – the approximation algorithm
- **scale** – a function to apply to the figure



---

## Module bn2graph

---

`pyAgrum.lib.bn2graph.BN2dot (bn, size='4', nodeColor=None, arcWidth=None, arcColor=None, cmapNode=None, cmapArc=None, showMsg=None)`  
 create a pydotplus representation of the BN

### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) –
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** – color map to show the vals of Nodes
- **cmapArc** – color map to show the vals of Arcs.
- **showMsg** – a nodeMap of values to be shown as tooltip

**Returns** the desired representation of the BN as a dot graph

`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 3)) –
- **size** (*string*) – size of the rendered graph
- **Inference engine** (*pyAgrum*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets. If targets={} then each node is a target

- **nodeColor** – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as bold arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** – color map to show the vals of Nodes
- **cmapArc** – color map to show the vals of Arcs

:param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

`pyAgrum.lib.bn2graph.dotize(aBN, name, format='pdf')`

From a bn, creates an image of the BN

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – 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.forDarkTheme()`

change the color for arcs and text in graphs to be more visible in dark theme

`pyAgrum.lib.bn2graph.forLightTheme()`

change the color for arcs and text in graphs to be more visible in light theme

`pyAgrum.lib.bn2graph.getBlackInTheme()`

return the color used for arc and text in graphs

`pyAgrum.lib.bn2graph.pdfize(aBN, name)`

From a bn, creates a pdf of the BN

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image

`pyAgrum.lib.bn2graph.pngize(aBN, name)`

From a bn, creates a png of the BN

#### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image

`pyAgrum.lib.bn2graph.proba2histo(p, scale=1.0)`

compute the representation of an histogram for a mono-dim Potential

**Parameters** **p** (`pyAgrum.Potential` (page 149)) – the mono-dim Potential

**Returns** a matplotlib histogram for a Potential p.

```
1 bn = gum.fastBN("a->b->d;a->c->d->e;f->b")
2 g = BNinference2dot(bn,
3     targets=['f', 'd'],
4     vals={'a': 1,
5           'b': 0.3,
6           'c': 0.3,
7           'd': 0.1,
8           'e': 0.1,
9           'f': 0.3},
10    arcvals={(0, 1): 2,
```

(continues on next page)

(continued from previous page)

```

11         (0, 2): 0.5})
12 g.write("test.png", format='png')

```

## 6.1 Visualization of Potentials

`pyAgrum.lib.bn2graph.proba2histo` (*p*, *scale=1.0*)  
compute the representation of an histogram for a mono-dim Potential

**Parameters** *p* (`pyAgrum.Potential` (page 149)) – the mono-dim Potential

**Returns** a matplotlib histogram for a Potential *p*.

## 6.2 Visualization of Bayesian Networks

`pyAgrum.lib.bn2graph.BN2dot` (*bn*, *size='4'*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmapNode=None*, *cmapArc=None*, *showMsg=None*)  
create a pydotplus representation of the BN

### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) –
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** – color map to show the vals of Nodes
- **cmapArc** – color map to show the vals of Arcs.
- **showMsg** – a nodeMap of values to be shown as tooltip

**Returns** the desired representation of the BN as a dot graph

`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 3)) –
- **size** (*string*) – size of the rendered graph
- **Inference engine** (*pyAgrum*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dictionary*) – map of evidence
- **targets** (*set*) – set of targets. If targets={} then each node is a target

- **nodeColor** – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
  - **arcWidth** – a arcMap of values to be shown as bold arcs
  - **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
  - **cmapNode** – color map to show the vals of Nodes
  - **cmapArc** – color map to show the vals of Arcs
- :param dag : only shows nodes that have their id in the dag (and not in the whole BN)

**Returns** the desired representation of the inference

## 6.3 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 3)) – 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 3)) – 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

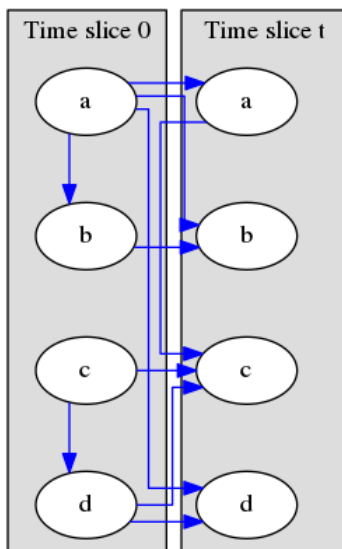
### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – the bayes net to show
- **name** (*string*) – the filename (without extension) for the image



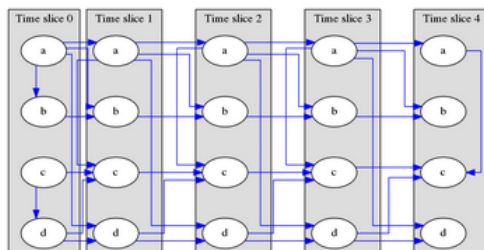
## Module dynamic bayesian network

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



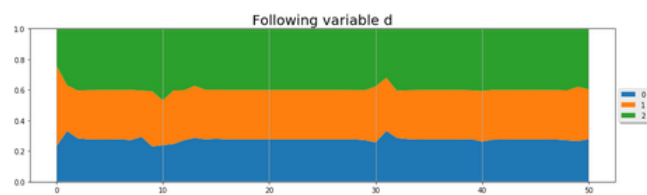
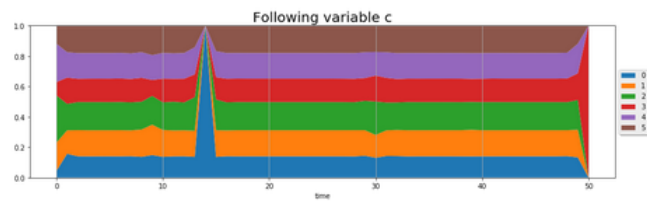
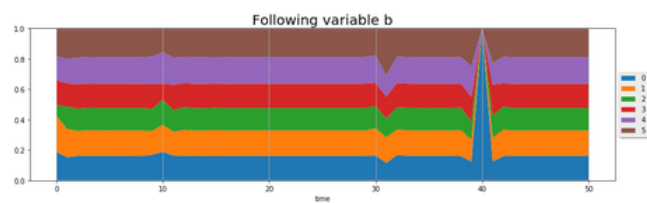
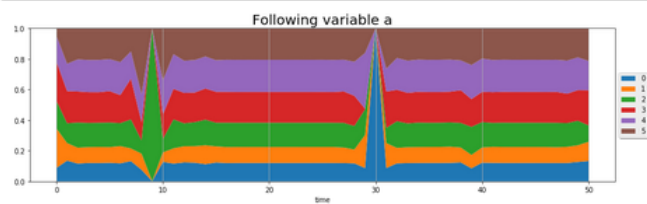
T=5

```
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, 'a30':0, 'c14':0, 'b40':0, 'c50':3})
```



Basic implementation for dynamic Bayesian Networks in pyAgrum

```
pyAgrum.lib.dynamicBN.getTimeSlices(dbn, size=None)
```

Try to correctly represent dBN and 2TBN as an HTML string

**Parameters**

- **dbn** – the dynamic BN
- **size** – size of the figure
- **format** – png/svg

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

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

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

**Returns** all the timeslice of a dbn

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

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

`pyAgrum.lib.dynamicBN.plotFollow (lovars, twoTdbn, T, evs)`

plots modifications of variables in a 2TDN knowing the size of the time window (T) and the evidence on the sequence.

**Parameters**

- **lovars** – list of variables to follow
- **twoTdbn** – the two-timeslice dbn
- **T** – the time range
- **evs** – observations

`pyAgrum.lib.dynamicBN.plotFollowUnrolled (lovars, dbn, T, evs)`

plot the dynamic evolution of a list of vars with a dBN

**Parameters**

- **lovars** – list of variables to follow
- **dbn** – the unrolled dbn
- **T** – the time range
- **evs** – observations

`pyAgrum.lib.dynamicBN.realNameFrom2TBNname (name, ts)`

@return dynamic name from static name and timeslice (no check)

`pyAgrum.lib.dynamicBN.showTimeSlices (dbn, size=None)`

Try to correctly display dBN and 2TBN

**Parameters**

- **dbn** – the dynamic BN
- **size** – size of the figure
- **format** – png/svg

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

unroll a 2TBN given the nbr of timeslices

**Parameters**

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

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

## 8.1 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.showROC` (*bn, csv\_name, variable, label, visible=True, show\_fig=False, with\_labels=True*)

Compute the ROC curve and save the result in the folder of the csv file.

### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – a bayesian network
- **csv\_name** (*str*) – a csv filename
- **target** (*str*) – the target
- **label** (*str*) – the target label
- **visible** (*bool*) – indicates if the resulting curve must be printed

## 8.2 bn2csv

Samples generation w.r.t to a probability distribution represented by a Bayesian network.

**class** `pyAgrum.lib.bn2csv.CSVGenerator`

Bases: `object`

Class for samples generation w.r.t to a probability distribution represented by a Bayesian network.

**caching\_probab** (*bn, node\_id, n, par*)

### Parameters

- **bn** (`pyAgrum.BayesNet` (page 3)) – a Bayesian network
- **node\_id** (*int*) – a node id
- **n** (*int*) – a node id
- **par** (*list*) – the node's parents

**Returns** the node's probabilities

**Return type** list

**cacheNameAndParents** (*bn, n*)

Compute a list of parents for node n in BN bn.

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 3)) – a Bayesian network
- **n** – (int) a node id
- **n** – (str) a node name

**Returns** a couple of name of n and list of parents names

**Return type** tuple

**static draw** (*tab*)

draw a value using tab as probability table.

**Parameters** **tab** (*list*) – a probability table

**Returns** the couple (i,proba)

**Return type** tuple

**static nameAndParents** (*bn, n*)

Compute a list of parents for node n in BN bn.

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 3)) – a Bayesian network
- **n** – (int) a node id
- **n** – (str) a node name

**Returns** a couple of name of n and list of parents names

**Return type** tuple

**Raises** `gum.IndexError` – If the node is not in the Bayesian network

**newSample** (*bn, seq*)

Generate a sample w.r.t to the bn using the variable sequence seq (topological order)

**Parameters**

- **bn** ([pyAgrum.BayesNet](#) (page 3)) – a Bayesian network
- **seq** (*list*) – a variable sequence

**Returns** the couple (sample,log2-likelihood)

**Return type** tuple

**proceed** (*name\_in, name\_out, n, visible, with\_labels*)

From the file name\_in (BN file), generate n samples and save them in name\_out

**Parameters**

- **name\_in** (*str*) – a file name
- **name\_out** (*str*) – the output file
- **n** (*int*) – the number of samples
- **visible** (*bool*) – indicate if a progress bar should be displayed
- **with\_labels** (*bool*) – indicate if values should be labelled or not

**Returns** the log2-likelihood of the n samples database

**Return type** double

`pyAgrum.lib.bn2csv.generateCSV(name_in, name_out, n, visible=False, with_labels=True)`

From the file `name_in` (BN file), generate `n` samples and save them in `name_out`

#### Parameters

- **name\_in** (*str*) – a file name
- **name\_out** (*str*) – the output file
- **n** (*int*) – the number of samples
- **visible** (*bool*) – indicate if a progress bar should be displayed
- **with\_labels** (*bool*) – indicate if values should be labelled or not

**Returns** the log2-likelihood of the `n` samples database

**Return type** double

`pyAgrum.lib.bn2csv.module_help(exit_value=1)`

defines help viewed if args are not OK on command line, and exit with `exit_value`

## 8.3 bn2scores

`pyAgrum.lib.bn2scores.checkCompatibility(bn, fields, csv_name)`

check if variables of the `bn` are in the `fields`

if not : return None if compatibility : 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)`

## 8.4 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 3)) – a BN or a filename for reference
- **name2** (*str* or `pyAgrum.BayesNet` (page 3)) – another BN or another 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 dictionary containing 'hamming', 'structural hamming'

**Return type** dict[double,double]

**scores ()**

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

precision and recall are computed considering BN1 as the reference

Fscore is  $2 * (\text{recall} * \text{precision}) / (\text{recall} + \text{precision})$  and is the weighted average of Precision and Recall.

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

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

**Return type** dict[str,double]

`pyAgrum.lib.bn_vs_bn.module_help (exit_value=1)`

defines help viewed if args are not OK on command line, and exit with exit\_value

## 8.5 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

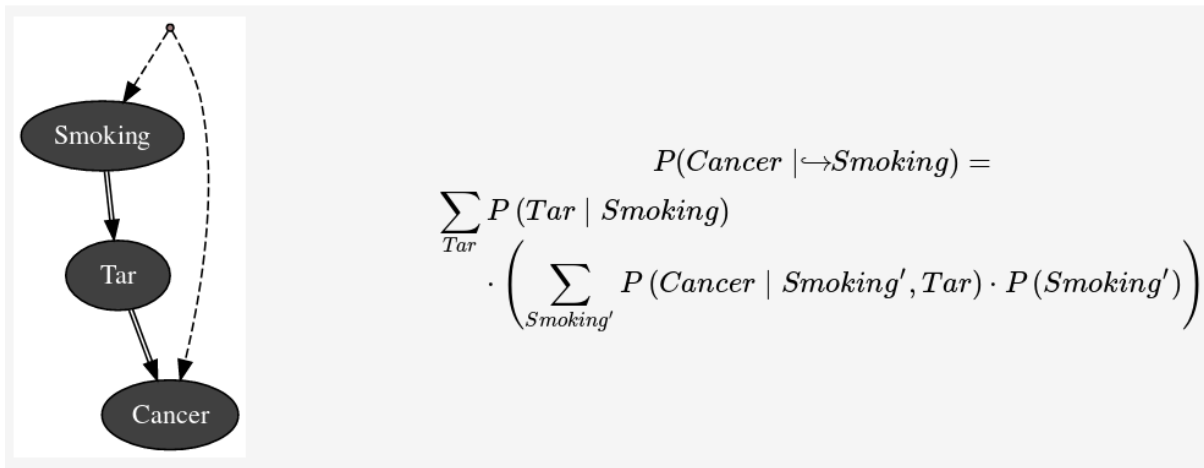
**Parameters** **cpt** – the Potential to represent

**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



Causality in pyAgrum mainly consists in the ability to build a causal model, i.e. a (observational) Bayesian network and a set of latent variables and their relation with observation variables and in the ability to compute using do-calculus the causal impact in such a model.

Causality is a set of pure python3 scripts based on pyAgrum's tools.

## 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 consists in

- a reference to the CausalModel
- Three sets of variables name that represent the 3 sets of variable in the query  $P(\text{set1} \mid \text{doing}(\text{set2}), \text{knowing}(\text{set3}))$ .
- the AST for compute the query.

```
class pyAgrum.causal.CausalFormula (cm: 'CausalModel', root: ASTtree, on: Union(str,
                                         NameSet), doing: Union(str, NameSet), knowing: Op-
                                         tional[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 :  $\$P(\text{on}|\text{knowing}, \text{overhook}(\text{doing}))\$$

**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 the AST. 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: Op-
                                tional[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) → pyA-
grum.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: pyA-  
                                         grum.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**

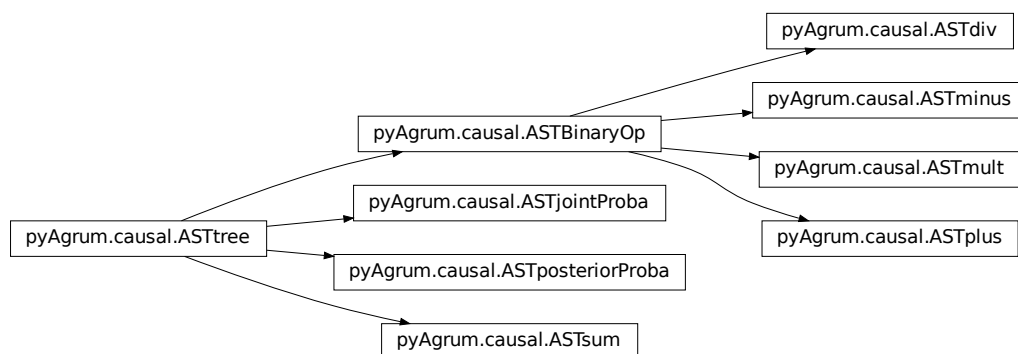
- **cm** – the causal model
- **Y** – The variables of interest (named following the paper)
- **X** – The variable of intervention (named following the paper)
- **P** – The ASTtree representing the calculus in construction

**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 182).



### 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

This file defines some helpers for handling causal concepts in notebooks

```

pyAgrum.causal.notebook.getCausalImpact (model: 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[str, pyAgrum.Potential, str]
    return a HTML representing of the three values defining a causal impact : formula, value, explanation
    :param model: the causal model :param on: the impacted variable(s) :param doing: the variable(s) of
    intervention :param knowing: the variable(s) of evidence :param values : values for certain variables

Returns a triplet (CausalFormula, gum.Potential, explanation)

pyAgrum.causal.notebook.getCausalModel (cm: pyAgrum.causal._CausalModel.CausalModel,
size=None) → str
    return a HTML representing the causal model :param cm: the causal model :param size: passd :param vals:
    :return:

pyAgrum.causal.notebook.showCausalImpact (model: 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)
    display a HTML representing of the three values defining a causal impact : formula, value, explanation
    :param model: the causal model :param on: the impacted variable(s) :param doing: the variable(s) of
    intervention :param knowing: the variable(s) of evidence :param values : values for certain variables

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

```





---

## Probabilistic Relational Models

---

For now, pyAgrum only allows to explore Probabilistic Relational Models written with o3prm syntax.

**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 149)

**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



## 11.1 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*) – the path to a BayesNet which contains lower probabilities.
- **src\_max\_den** (*str*) – the (optional) path to a BayesNet which contains upper probabilities.

**CredalNet(src\_min\_num,src\_max\_den)** -> **CredalNet**

**Parameters:**

- **\*\*src\_min\_num** (*pyAgrum.BayesNet*) – the BayesNet which contains lower probabilities.
- **\*\*src\_max\_den** (*pyAgrum.BayesNet*) – the (optional) BayesNet 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 original credal sets and may induce huge imprecision.

**Warning:** Enlarge the original credal sets and therefor induce huge imprecision by propagation. Not recommended, use MCSampling or something else instead

**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 is provided with counts, to False if two BayesNets are provided; one with probabilities (the lower net) and one with denominators over the first modalities (the upper net)
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept

**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 chosen NodeId

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



**Return type** *pyAgrum.CredalNet* (page 195)

**current\_bn** (*CredalNet self*)

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

**Return type** *pyAgrum.BayesNet* (page 3)

**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 144)) – 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 144)

**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.

Does 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 195)

**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 144)) – 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 3)

## 11.2 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  $max \leq 1$

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

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

**Raises** `gum.OutOfLowerBound` – If  $timeout \leq 0.0$

**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 inf*)

`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** **timeout** (*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`



### 12.1 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 131)) – 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 131)) – 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 131)) – 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 131)) – 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

**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

**completeInstantiation** (*DAGmodel self*)

Get an instantiation over all the variables of the model.

**Returns** the complete instantiation

**Return type** `pyAgrum.instantiation`

**cpt** (*InfluenceDiagram self, int varId*)

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 149)

**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 117)

**decisionNodeSize** (*InfluenceDiagram self*)

**Returns** the number of decision nodes

**Return type** int

**decisionOrderExists** (*InfluenceDiagram self*)

**Returns** True if a directed path exist with all decision node

**Return type** bool

**empty** (*DAGmodel self*)

**Returns** True if the model is empty

**Return type** bool

**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 131)) – 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 113)) – The arc to be removed.
- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

**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

**getDecisionGraph** (*InfluenceDiagram self*)

**Returns** the temporal Graph.

**Return type** *pyAgrum.DAG* (page 117)

**getDecisionOrder** (*InfluenceDiagram self*)

**Returns** the sequence of decision nodes in the directed path.

**Return type** list

**Raises** *NotFound* (page 228) – If such a path does not exist

**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** ()

Deprecated method in pyAgrum>0.12.0. See nodes instead.

**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

**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.IOException` – If file not found
- `gum.FatalError` – If file is not valid

**log10DomainSize** (*DAGmodel self*)

**Returns** The log10 domain size of the joint probability for the model.

**Return type** double

**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 119)

**names** (*InfluenceDiagram self*)

**Returns** The names of the InfluenceDiagram variables

**Return type** list

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

**Parameters** **var** ([pyAgrum.DiscreteVariable](#) (page 131)) – 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

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

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

**Returns** the set of the parents ids.

**Return type** set

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

**Warning:** Unreferenced function

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

**Warning:** Unreferenced function

**saveBIFXML** (*InfluenceDiagram self, str name*)

Save the BayesNet in a BIFXML file.

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

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

**Warning:** Unreferenced function

**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*)

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



**Returns** the utility table of the node

**Return type** [pyAgrum.Potential](#) (page 149)

**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 variable given its node id

**Return type** [pyAgrum.DiscreteVariable](#) (page 131)

**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 131)

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

**variableNodeMap** (*DAGmodel self*)

**Returns** the variable node map

**Return type** `pyAgrum.variableNodeMap`

## 12.2 Inference

**class** `pyAgrum.InfluenceDiagramInference` (*infDiag: pyAgrum.InfluenceDiagram*)

Proxy of C++ `pyAgrum.InfluenceDiagramInference` class. Proxy of C++ `pyAgrum.InfluenceDiagramInference` class.

**displayResult** (*InfluenceDiagramInference self*)

Displays the result of an inference.

**displayStrongJunctionTree** (*InfluenceDiagramInference self, ostream stream=cout*)

Displays on terminal the result of strong junction tree computation for test purpose only.

**Parameters** `args` (*TBW*) –

**eraseAllEvidence** (*InfluenceDiagramInference self*)

Removes all the evidence entered into the diagram.

**eraseEvidence** (*InfluenceDiagramInference self, Potential evidence*)

**Parameters** `evidence` ([pyAgrum.Potential](#) (page 149)) – the evidence to remove

**Raises** `gum.IndexError` – If the evidence does not belong to the influence diagram

**getBestDecisionChoice** (*InfluenceDiagramInference self, int decisionId*)

Returns best choice for decision variable given in parameter ( based upon MEU criteria )

**Parameters** `decisionId` (*int*) – the id of the decision variable

**Raises**

- `gum.OperationNotAllowed` – If no inference have yet been made
- `gum.InvalidNode` – If node given in parameter is not a decision node

**getMEU** (*InfluenceDiagramInference self*)

Returns maximum expected utility obtained from inference.

**Raises** `gum.OperationNotAllowed` – If no inference have yet been made

**influenceDiagram** (*InfluenceDiagramInference 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 207)

**insertEvidence** (*InfluenceDiagramInference self, pyAgrum.List< pyAgrum.Potential \* > evidenceList*)

Insert new evidence in the graph.

**Parameters** **evidenceList** (*list*) – a list of potentials as evidences

**Warning:** If an evidence already w.r.t. a given node and a new evidence w.r.t. this node is onserted, the old evidence is removed

**Raises** `gum.OperationNotAllowed` – If an evidence is over more than one variable

**junctionTreeToDot** (*InfluenceDiagramInference self*)

**Returns** the result of strong junction tree computation for test purpose only.

**Return type** `str`

**makeInference** (*InfluenceDiagramInference self*)

Makes the inference.

**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

---

## Functions from pyAgrum

---

Useful functions in pyAgrum

`pyAgrum.about()`  
about() for pyAgrum

`pyAgrum.fastBN(arcs, domain_size=2)`  
rapid prototyping of BN.

### Parameters

- **arcs** – dot-like simple list of arcs (“a->b->c;a->c->d” for instance). The first apparition of a node name can be enhanced with a “[domain\_size]” extension. For instance “a[5]->b->c;a[2]->c->d” will create a BN with a variable “a” whos domain size is `a.nbrDim()==5` (the second “a[2]” is not taken into account since the variable has already been created).
- **domain\_size** – the domain size of each created variable.

**Returns** the created `pyAgrum.BayesNet`

`pyAgrum.getPosterior(bn, 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 3)) –
- **evs** (*dictionary*) – events map {name/id:val, name/id : [ val1, val2 ], ... }
- **target** – variable name or id

**Returns** posterior Potential

## 13.1 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 file with optional listeners and arguments

**Parameters**

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

**Returns** a BN from a file using one of the availableBNExts() suffixes.

Listeners could be added in order to monitor its loading.

## Examples

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

**Parma** `bn(gum.BayesNet)` the BN to save

**Parameters** **filename (str)** – the name of the output file

## 13.2 Input for influence diagram

`pyAgrum.loadID(filename)`  
read a `gum.InfluenceDiagram` from a `bifxml` file

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

**Returns** an `InfluenceDiagram`

---

Other functions from aGrUM

---

## 14.1 Listeners

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

### 14.1.1 LoadListener

Listeners could be added in order to monitor the progress when loading a pyAgrum.BayesNet

```
>>> import pyAgrum as gum
>>>
>>> # creating a new listeners
>>> def foo(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>>
>>> gum.loadBN('./bn.bif',listeners=[foo,bar])
>>> # .....#.....#.....#.....#...50%
>>> # .....#.....#.....#.....#.....#.....% | bn loaded
```

### 14.1.2 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
```

### 14.1.3 ApproximationSchemeListener

### 14.1.4 DatabaseGenerationListener

## 14.2 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.

## 14.3 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





---

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

Proxy of C++ pyAgrim.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** pyAgrim.**FatalError** (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.**FormatNotFound** (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.InvalidArc (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.InvalidArgument (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.InvalidArgumentsNumber (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.**SyntaxError** (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.**UndefinedElement** (\*args)

Proxy of C++ pyAgrim.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** pyAgrim.**UndefinedIteratorKey** (\*args)

Proxy of C++ pyAgrim.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 16

---

### Indices and tables

---

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



### p

`pyAgrum.causal.notebook`, [186](#)  
`pyAgrum.lib.bn2graph`, [169](#)  
`pyAgrum.lib.notebook`, [157](#)





## A

- `about()` (in module *pyAgrum*), 215
- `abs()` (*pyAgrum.Potential* method), 149
- `add()` (*pyAgrum.BayesNet* method), 4
- `add()` (*pyAgrum.InfluenceDiagram* method), 207
- `add()` (*pyAgrum.Instantiation* method), 144
- `add()` (*pyAgrum.Potential* method), 149
- `addAllTargets()` (*pyAgrum.GibbsSampling* method), 56
- `addAllTargets()` (*pyAgrum.ImportanceSampling* method), 75
- `addAllTargets()` (*pyAgrum.LazyPropagation* method), 31
- `addAllTargets()` (*pyAgrum.LoopyBeliefPropagation* method), 50
- `addAllTargets()` (*pyAgrum.LoopyGibbsSampling* method), 81
- `addAllTargets()` (*pyAgrum.LoopyImportanceSampling* method), 100
- `addAllTargets()` (*pyAgrum.LoopyMonteCarloSampling* method), 87
- `addAllTargets()` (*pyAgrum.LoopyWeightedSampling* method), 93
- `addAllTargets()` (*pyAgrum.MonteCarloSampling* method), 63
- `addAllTargets()` (*pyAgrum.ShaferShenoyInference* method), 38
- `addAllTargets()` (*pyAgrum.VariableElimination* method), 44
- `addAllTargets()` (*pyAgrum.WeightedSampling* method), 69
- `addAMPLITUDE()` (*pyAgrum.BayesNet* method), 4
- `addAND()` (*pyAgrum.BayesNet* method), 4
- `addArc()` (*pyAgrum.BayesNet* method), 4
- `addArc()` (*pyAgrum.CredalNet* method), 195
- `addArc()` (*pyAgrum.DAG* method), 117
- `addArc()` (*pyAgrum.DiGraph* method), 115
- `addArc()` (*pyAgrum.InfluenceDiagram* method), 207
- `addArc()` (*pyAgrum.MixedGraph* method), 126
- `addChanceNode()` (*pyAgrum.InfluenceDiagram* method), 208
- `addCOUNT()` (*pyAgrum.BayesNet* method), 5
- `addDecisionNode()` (*pyAgrum.InfluenceDiagram* method), 208
- `addEdge()` (*pyAgrum.CliqueGraph* method), 122
- `addEdge()` (*pyAgrum.MixedGraph* method), 126
- `addEdge()` (*pyAgrum.UndiGraph* method), 120
- `addEvidence()` (*pyAgrum.GibbsSampling* method), 56
- `addEvidence()` (*pyAgrum.ImportanceSampling* method), 75
- `addEvidence()` (*pyAgrum.LazyPropagation* method), 31
- `addEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 50
- `addEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 81
- `addEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 100
- `addEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 87
- `addEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 93
- `addEvidence()` (*pyAgrum.MonteCarloSampling* method), 63
- `addEvidence()` (*pyAgrum.ShaferShenoyInference* method), 38
- `addEvidence()` (*pyAgrum.VariableElimination* method), 44
- `addEvidence()` (*pyAgrum.WeightedSampling* method), 69
- `addEXISTS()` (*pyAgrum.BayesNet* method), 5
- `addFORALL()` (*pyAgrum.BayesNet* method), 5
- `addForbiddenArc()` (*pyAgrum.BN Learner* method), 106
- `addJointTarget()` (*pyAgrum.LazyPropagation* method), 31
- `addJointTarget()` (*pyAgrum.ShaferShenoyInference* method),

- 38
- `addJointTarget()` (*pyAgrum.VariableElimination* method), 45
- `addLabel()` (*pyAgrum.LabelizedVariable* method), 134
- `addLogit()` (*pyAgrum.BayesNet* method), 5
- `addMandatoryArc()` (*pyAgrum.BN Learner* method), 106
- `addMAX()` (*pyAgrum.BayesNet* method), 5
- `addMEDIAN()` (*pyAgrum.BayesNet* method), 6
- `addMIN()` (*pyAgrum.BayesNet* method), 6
- `addNode()` (*pyAgrum.CliqueGraph* method), 122
- `addNode()` (*pyAgrum.DAG* method), 117
- `addNode()` (*pyAgrum.DiGraph* method), 115
- `addNode()` (*pyAgrum.MixedGraph* method), 126
- `addNode()` (*pyAgrum.UndiGraph* method), 120
- `addNodes()` (*pyAgrum.CliqueGraph* method), 122
- `addNodes()` (*pyAgrum.DAG* method), 118
- `addNodes()` (*pyAgrum.DiGraph* method), 115
- `addNodes()` (*pyAgrum.MixedGraph* method), 126
- `addNodes()` (*pyAgrum.UndiGraph* method), 120
- `addNodeWithId()` (*pyAgrum.CliqueGraph* method), 122
- `addNodeWithId()` (*pyAgrum.DAG* method), 118
- `addNodeWithId()` (*pyAgrum.DiGraph* method), 115
- `addNodeWithId()` (*pyAgrum.MixedGraph* method), 126
- `addNodeWithId()` (*pyAgrum.UndiGraph* method), 120
- `addNoisyAND()` (*pyAgrum.BayesNet* method), 6
- `addNoisyOR()` (*pyAgrum.BayesNet* method), 6
- `addNoisyORCompound()` (*pyAgrum.BayesNet* method), 6
- `addNoisyORNet()` (*pyAgrum.BayesNet* method), 7
- `addOR()` (*pyAgrum.BayesNet* method), 7
- `addPossibleEdge()` (*pyAgrum.BN Learner* method), 106
- `addStructureListener()` (*pyAgrum.BayesNet* method), 7
- `addStructureListener()` (*pyAgrum.BayesNet Fragment* method), 25
- `addTarget()` (*pyAgrum.GibbsSampling* method), 57
- `addTarget()` (*pyAgrum.ImportanceSampling* method), 75
- `addTarget()` (*pyAgrum.LazyPropagation* method), 32
- `addTarget()` (*pyAgrum.LoopyBeliefPropagation* method), 51
- `addTarget()` (*pyAgrum.LoopyGibbsSampling* method), 81
- `addTarget()` (*pyAgrum.LoopyImportanceSampling* method), 100
- `addTarget()` (*pyAgrum.LoopyMonteCarloSampling* method), 88
- `addTarget()` (*pyAgrum.LoopyWeightedSampling* method), 94
- `addTarget()` (*pyAgrum.MonteCarloSampling* method), 63
- `addTarget()` (*pyAgrum.ShaferShenoyInference* method), 38
- `addTarget()` (*pyAgrum.VariableElimination* method), 45
- `addTarget()` (*pyAgrum.WeightedSampling* method), 69
- `addTick()` (*pyAgrum.DiscretizedVariable* method), 136
- `addToClique()` (*pyAgrum.CliqueGraph* method), 122
- `addUtilityNode()` (*pyAgrum.InfluenceDiagram* method), 208
- `addVariable()` (*pyAgrum.CredalNet* method), 195
- `addWeightedArc()` (*pyAgrum.BayesNet* method), 8
- `aggType` (*pyAgrum.PRMexplorer* attribute), 189
- `animApproximationScheme()` (in module *pyAgrum.lib.notebook*), 157, 167
- `approximatedBinarization()` (*pyAgrum.CredalNet* method), 196
- `Arc` (class in *pyAgrum*), 113
- `arcs()` (*pyAgrum.BayesNet* method), 8
- `arcs()` (*pyAgrum.BayesNet Fragment* method), 25
- `arcs()` (*pyAgrum.DAG* method), 118
- `arcs()` (*pyAgrum.DiGraph* method), 115
- `arcs()` (*pyAgrum.EssentialGraph* method), 22
- `arcs()` (*pyAgrum.InfluenceDiagram* method), 209
- `arcs()` (*pyAgrum.MarkovBlanket* method), 23
- `arcs()` (*pyAgrum.MixedGraph* method), 126
- `argmax()` (*pyAgrum.Potential* method), 149
- `argmin()` (*pyAgrum.Potential* method), 149
- `ASTBinaryOp` (class in *pyAgrum.causal*), 183
- `ASTdiv` (class in *pyAgrum.causal*), 184
- `ASTjointProba` (class in *pyAgrum.causal*), 185
- `ASTminus` (class in *pyAgrum.causal*), 184
- `ASTmult` (class in *pyAgrum.causal*), 184
- `ASTplus` (class in *pyAgrum.causal*), 183
- `ASTposteriorProba` (class in *pyAgrum.causal*), 185
- `ASTsum` (class in *pyAgrum.causal*), 185
- `ASTtree` (class in *pyAgrum.causal*), 182
- `availableBNExts()` (in module *pyAgrum*), 215
- ## B
- `BayesNet` (class in *pyAgrum*), 3
- `BayesNet Fragment` (class in *pyAgrum*), 24
- `beginTopologyTransformation()` (*pyAgrum.BayesNet* method), 8
- `belongs()` (*pyAgrum.RangeVariable* method), 139
- `binaryJoinTree()` (*pyAgrum.JunctionTreeGenerator* method), 21
- `bn` (*pyAgrum.causal.ASTposteriorProba* attribute), 186
- `BN()` (*pyAgrum.GibbsSampling* method), 56

- BN() (*pyAgrum.ImportanceSampling method*), 74  
 BN() (*pyAgrum.LazyPropagation method*), 30  
 BN() (*pyAgrum.LoopyBeliefPropagation method*), 50  
 BN() (*pyAgrum.LoopyGibbsSampling method*), 81  
 BN() (*pyAgrum.LoopyImportanceSampling method*), 99  
 BN() (*pyAgrum.LoopyMonteCarloSampling method*), 87  
 BN() (*pyAgrum.LoopyWeightedSampling method*), 93  
 BN() (*pyAgrum.MonteCarloSampling method*), 62  
 BN() (*pyAgrum.ShaferShenoyInference method*), 37  
 BN() (*pyAgrum.VariableElimination method*), 44  
 BN() (*pyAgrum.WeightedSampling method*), 68  
 BN2dot() (*in module pyAgrum.lib.bn2graph*), 169, 171  
 BNDatabaseGenerator (*class in pyAgrum*), 16  
 BNinference2dot() (*in module pyAgrum.lib.bn2graph*), 169, 171  
 BNLearner (*class in pyAgrum*), 105  
 bnToCredal() (*pyAgrum.CredalNet method*), 196  
 burnIn() (*pyAgrum.GibbsBNdistance method*), 18  
 burnIn() (*pyAgrum.GibbsSampling method*), 57  
 burnIn() (*pyAgrum.LoopyGibbsSampling method*), 81
- ## C
- causalBN() (*pyAgrum.causal.CausalModel method*), 180  
 CausalFormula (*class in pyAgrum.causal*), 180  
 causalImpact() (*in module pyAgrum.causal*), 181  
 CausalModel (*class in pyAgrum.causal*), 179  
 chanceNodeSize() (*pyAgrum.InfluenceDiagram method*), 209  
 changeLabel() (*pyAgrum.LabelizedVariable method*), 134  
 changePotential() (*pyAgrum.BayesNet method*), 8  
 changeVariableLabel() (*pyAgrum.BayesNet method*), 8  
 changeVariableName() (*pyAgrum.BayesNet method*), 8  
 changeVariableName() (*pyAgrum.InfluenceDiagram method*), 209  
 checkConsistency() (*pyAgrum.BayesNetFragment method*), 25  
 chgEvidence() (*pyAgrum.GibbsSampling method*), 57  
 chgEvidence() (*pyAgrum.ImportanceSampling method*), 75  
 chgEvidence() (*pyAgrum.LazyPropagation method*), 32  
 chgEvidence() (*pyAgrum.LoopyBeliefPropagation method*), 51  
 chgEvidence() (*pyAgrum.LoopyGibbsSampling method*), 82  
 chgEvidence() (*pyAgrum.LoopyImportanceSampling method*), 100  
 chgEvidence() (*pyAgrum.LoopyMonteCarloSampling method*), 88  
 chgEvidence() (*pyAgrum.LoopyWeightedSampling method*), 94  
 chgEvidence() (*pyAgrum.MonteCarloSampling method*), 63  
 chgEvidence() (*pyAgrum.ShaferShenoyInference method*), 39  
 chgEvidence() (*pyAgrum.VariableElimination method*), 45  
 chgEvidence() (*pyAgrum.WeightedSampling method*), 69  
 chgVal() (*pyAgrum.Instantiation method*), 144  
 chi2() (*pyAgrum.BNLearner method*), 106  
 children() (*pyAgrum.BayesNet method*), 9  
 children() (*pyAgrum.BayesNetFragment method*), 25  
 children() (*pyAgrum.causal.CausalModel method*), 180  
 children() (*pyAgrum.DAG method*), 118  
 children() (*pyAgrum.DiGraph method*), 115  
 children() (*pyAgrum.EssentialGraph method*), 22  
 children() (*pyAgrum.InfluenceDiagram method*), 209  
 children() (*pyAgrum.MarkovBlanket method*), 23  
 children() (*pyAgrum.MixedGraph method*), 127  
 classAggregates() (*pyAgrum.PRMexplorer method*), 189  
 classAttributes() (*pyAgrum.PRMexplorer method*), 189  
 classDag() (*pyAgrum.PRMexplorer method*), 189  
 classes() (*pyAgrum.PRMexplorer method*), 190  
 classImplements() (*pyAgrum.PRMexplorer method*), 189  
 classParameters() (*pyAgrum.PRMexplorer method*), 190  
 classReferences() (*pyAgrum.PRMexplorer method*), 190  
 classSlotChains() (*pyAgrum.PRMexplorer method*), 190  
 clear() (*pyAgrum.CliqueGraph method*), 123  
 clear() (*pyAgrum.DAG method*), 118  
 clear() (*pyAgrum.DiGraph method*), 115  
 clear() (*pyAgrum.Instantiation method*), 145  
 clear() (*pyAgrum.MixedGraph method*), 127  
 clear() (*pyAgrum.UndiGraph method*), 120  
 clearEdges() (*pyAgrum.CliqueGraph method*), 123  
 clique() (*pyAgrum.CliqueGraph method*), 123  
 CliqueGraph (*class in pyAgrum*), 122  
 cm (*pyAgrum.causal.CausalFormula attribute*), 180  
 CNLoopyPropagation (*class in pyAgrum*), 202  
 CNMonteCarloSampling (*class in pyAgrum*), 200  
 col() (*pyAgrum.SyntaxError method*), 231  
 completeInstantiation() (*pyAgrum.BayesNet method*), 9

`completeInstantiation()` (*pyAgrum.BayesNetFragment method*), 25  
`completeInstantiation()` (*pyAgrum.InfluenceDiagram method*), 209  
`compute()` (*pyAgrum.ExactBNdistance method*), 17  
`compute()` (*pyAgrum.GibbsBNdistance method*), 18  
`computeCPTMinMax()` (*pyAgrum.CredalNet method*), 196  
`configuration()` (in module *pyAgrum.lib.notebook*), 157, 163  
`connectedComponents()` (*pyAgrum.BayesNet method*), 9  
`connectedComponents()` (*pyAgrum.BayesNetFragment method*), 25  
`connectedComponents()` (*pyAgrum.CliqueGraph method*), 123  
`connectedComponents()` (*pyAgrum.DAG method*), 118  
`connectedComponents()` (*pyAgrum.DiGraph method*), 115  
`connectedComponents()` (*pyAgrum.EssentialGraph method*), 22  
`connectedComponents()` (*pyAgrum.MarkovBlanket method*), 23  
`connectedComponents()` (*pyAgrum.MixedGraph method*), 127  
`connectedComponents()` (*pyAgrum.UndiGraph method*), 120  
`container()` (*pyAgrum.CliqueGraph method*), 123  
`containerPath()` (*pyAgrum.CliqueGraph method*), 123  
`contains()` (*pyAgrum.Instantiation method*), 145  
`contains()` (*pyAgrum.Potential method*), 149  
`continueApproximationScheme()` (*pyAgrum.GibbsBNdistance method*), 18  
`copy()` (*pyAgrum.causal.ASTBinaryOp method*), 183  
`copy()` (*pyAgrum.causal.ASTdiv method*), 184  
`copy()` (*pyAgrum.causal.ASTjointProba method*), 185  
`copy()` (*pyAgrum.causal.ASTminus method*), 184  
`copy()` (*pyAgrum.causal.ASTmult method*), 184  
`copy()` (*pyAgrum.causal.ASTplus method*), 183  
`copy()` (*pyAgrum.causal.ASTposteriorProba method*), 186  
`copy()` (*pyAgrum.causal.ASTsum method*), 185  
`copy()` (*pyAgrum.causal.ASTtree method*), 182  
`copy()` (*pyAgrum.causal.CausalFormula method*), 180  
`cpf()` (*pyAgrum.PRMexplorer method*), 190  
`cpt()` (*pyAgrum.BayesNet method*), 9  
`cpt()` (*pyAgrum.BayesNetFragment method*), 26  
`cpt()` (*pyAgrum.InfluenceDiagram method*), 209  
`CredalNet` (class in *pyAgrum*), 195  
`credalNet_currentCpt()` (*pyAgrum.CredalNet method*), 196  
`credalNet_srcCpt()` (*pyAgrum.CredalNet method*), 196  
`current_bn()` (*pyAgrum.CredalNet method*), 197  
`currentNodeType()` (*pyAgrum.CredalNet method*), 196  
`currentPosterior()` (*pyAgrum.GibbsSampling method*), 57  
`currentPosterior()` (*pyAgrum.ImportanceSampling method*), 76  
`currentPosterior()` (*pyAgrum.LoopyGibbsSampling method*), 82  
`currentPosterior()` (*pyAgrum.LoopyImportanceSampling method*), 101  
`currentPosterior()` (*pyAgrum.LoopyMonteCarloSampling method*), 88  
`currentPosterior()` (*pyAgrum.LoopyWeightedSampling method*), 95  
`currentPosterior()` (*pyAgrum.MonteCarloSampling method*), 64  
`currentPosterior()` (*pyAgrum.WeightedSampling method*), 70  
`currentTime()` (*pyAgrum.BNlearner method*), 107  
`currentTime()` (*pyAgrum.CNLoopyPropagation method*), 203  
`currentTime()` (*pyAgrum.CNMonteCarloSampling method*), 200  
`currentTime()` (*pyAgrum.GibbsBNdistance method*), 18  
`currentTime()` (*pyAgrum.GibbsSampling method*), 58  
`currentTime()` (*pyAgrum.ImportanceSampling method*), 76  
`currentTime()` (*pyAgrum.LoopyBeliefPropagation method*), 51  
`currentTime()` (*pyAgrum.LoopyGibbsSampling method*), 82  
`currentTime()` (*pyAgrum.LoopyImportanceSampling method*), 101  
`currentTime()` (*pyAgrum.LoopyMonteCarloSampling method*), 89  
`currentTime()` (*pyAgrum.LoopyWeightedSampling method*), 95  
`currentTime()` (*pyAgrum.MonteCarloSampling method*), 64  
`currentTime()` (*pyAgrum.WeightedSampling method*), 70

## D

`DAG` (class in *pyAgrum*), 117  
`dag()` (*pyAgrum.BayesNet method*), 9  
`dag()` (*pyAgrum.BayesNetFragment method*), 26  
`dag()` (*pyAgrum.InfluenceDiagram method*), 209  
`dag()` (*pyAgrum.MarkovBlanket method*), 24



- database() (pyAgrum.BNDatabaseGenerator method), 16  
 databaseWeight() (pyAgrum.BN Learner method), 107  
 dec() (pyAgrum.Instantiation method), 145  
 decIn() (pyAgrum.Instantiation method), 145  
 decisionNodeSize() (pyAgrum.InfluenceDiagram method), 209  
 decisionOrderExists() (pyAgrum.InfluenceDiagram method), 210  
 decNotVar() (pyAgrum.Instantiation method), 145  
 decOut() (pyAgrum.Instantiation method), 145  
 decVar() (pyAgrum.Instantiation method), 145  
 DefaultInLabel, 221  
 description() (pyAgrum.DiscreteVariable method), 131  
 description() (pyAgrum.DiscretizedVariable method), 136  
 description() (pyAgrum.LabelizedVariable method), 134  
 description() (pyAgrum.RangeVariable method), 139  
 DiGraph (class in pyAgrum), 114  
 dim() (pyAgrum.BayesNet method), 10  
 dim() (pyAgrum.BayesNetFragment method), 26  
 disableEpsilon() (pyAgrum.GibbsBNdistance method), 18  
 disableMaxIter() (pyAgrum.GibbsBNdistance method), 18  
 disableMaxTime() (pyAgrum.GibbsBNdistance method), 18  
 disableMinEpsilonRate() (pyAgrum.GibbsBNdistance method), 18  
 DiscreteVariable (class in pyAgrum), 131  
 DiscretizedVariable (class in pyAgrum), 136  
 displayResult() (pyAgrum.InfluenceDiagramInference method), 213  
 displayStrongJunctionTree() (pyAgrum.InfluenceDiagramInference method), 213  
 doCalculusWithObservation() (in module pyAgrum.causal), 181  
 domain() (pyAgrum.DiscreteVariable method), 131  
 domain() (pyAgrum.DiscretizedVariable method), 136  
 domain() (pyAgrum.LabelizedVariable method), 134  
 domain() (pyAgrum.RangeVariable method), 139  
 domainSize() (pyAgrum.CredalNet method), 197  
 domainSize() (pyAgrum.DiscreteVariable method), 131  
 domainSize() (pyAgrum.DiscretizedVariable method), 136  
 domainSize() (pyAgrum.Instantiation method), 145  
 domainSize() (pyAgrum.LabelizedVariable method), 134  
 domainSize() (pyAgrum.RangeVariable method), 139  
 dotize() (in module pyAgrum.lib.bn2graph), 170, 172  
 draw() (pyAgrum.Potential method), 149  
 drawSamples() (pyAgrum.BNDatabaseGenerator method), 16  
 DuplicateElement, 221  
 DuplicateLabel, 222  
 dynamicExpMax() (pyAgrum.CNLoopyPropagation method), 203  
 dynamicExpMax() (pyAgrum.CNMonteCarloSampling method), 200  
 dynamicExpMin() (pyAgrum.CNLoopyPropagation method), 203  
 dynamicExpMin() (pyAgrum.CNMonteCarloSampling method), 200
- ## E
- Edge (class in pyAgrum), 114  
 edges() (pyAgrum.CliqueGraph method), 123  
 edges() (pyAgrum.EssentialGraph method), 22  
 edges() (pyAgrum.MixedGraph method), 127  
 edges() (pyAgrum.UndiGraph method), 120  
 eliminationOrder() (pyAgrum.JunctionTreeGenerator method), 21  
 empty() (pyAgrum.BayesNet method), 10  
 empty() (pyAgrum.BayesNetFragment method), 26  
 empty() (pyAgrum.CliqueGraph method), 124  
 empty() (pyAgrum.DAG method), 118  
 empty() (pyAgrum.DiGraph method), 115  
 empty() (pyAgrum.DiscreteVariable method), 131  
 empty() (pyAgrum.DiscretizedVariable method), 137  
 empty() (pyAgrum.InfluenceDiagram method), 210  
 empty() (pyAgrum.Instantiation method), 145  
 empty() (pyAgrum.LabelizedVariable method), 134  
 empty() (pyAgrum.MixedGraph method), 127  
 empty() (pyAgrum.Potential method), 149  
 empty() (pyAgrum.RangeVariable method), 139  
 empty() (pyAgrum.UndiGraph method), 120  
 emptyArcs() (pyAgrum.DAG method), 118  
 emptyArcs() (pyAgrum.DiGraph method), 116  
 emptyArcs() (pyAgrum.MixedGraph method), 127  
 EmptyBSTree, 222  
 emptyEdges() (pyAgrum.CliqueGraph method), 124  
 emptyEdges() (pyAgrum.MixedGraph method), 127  
 emptyEdges() (pyAgrum.UndiGraph method), 121  
 EmptySet, 222  
 enableEpsilon() (pyAgrum.GibbsBNdistance method), 18  
 enableMaxIter() (pyAgrum.GibbsBNdistance method), 18

`enableMaxTime()` (*pyAgrum.GibbsBNdistance method*), 18  
`enableMinEpsilonRate()` (*pyAgrum.GibbsBNdistance method*), 18  
`end()` (*pyAgrum.Instantiation method*), 145  
`endTopologyTransformation()` (*pyAgrum.BayesNet method*), 10  
`entropy()` (*pyAgrum.Potential method*), 149  
`epsilon()` (*pyAgrum.BNLearner method*), 107  
`epsilon()` (*pyAgrum.CNLoopyPropagation method*), 203  
`epsilon()` (*pyAgrum.CNMonteCarloSampling method*), 201  
`epsilon()` (*pyAgrum.GibbsBNdistance method*), 18  
`epsilon()` (*pyAgrum.GibbsSampling method*), 58  
`epsilon()` (*pyAgrum.ImportanceSampling method*), 76  
`epsilon()` (*pyAgrum.LoopyBeliefPropagation method*), 51  
`epsilon()` (*pyAgrum.LoopyGibbsSampling method*), 82  
`epsilon()` (*pyAgrum.LoopyImportanceSampling method*), 101  
`epsilon()` (*pyAgrum.LoopyMonteCarloSampling method*), 89  
`epsilon()` (*pyAgrum.LoopyWeightedSampling method*), 95  
`epsilon()` (*pyAgrum.MonteCarloSampling method*), 64  
`epsilon()` (*pyAgrum.WeightedSampling method*), 70  
`epsilonMax()` (*pyAgrum.CredalNet method*), 197  
`epsilonMean()` (*pyAgrum.CredalNet method*), 197  
`epsilonMin()` (*pyAgrum.CredalNet method*), 197  
`erase()` (*pyAgrum.BayesNet method*), 10  
`erase()` (*pyAgrum.InfluenceDiagram method*), 210  
`erase()` (*pyAgrum.Instantiation method*), 146  
`eraseAllEvidence()` (*pyAgrum.CNLoopyPropagation method*), 203  
`eraseAllEvidence()` (*pyAgrum.GibbsSampling method*), 58  
`eraseAllEvidence()` (*pyAgrum.ImportanceSampling method*), 76  
`eraseAllEvidence()` (*pyAgrum.InfluenceDiagramInference method*), 213  
`eraseAllEvidence()` (*pyAgrum.LazyPropagation method*), 32  
`eraseAllEvidence()` (*pyAgrum.LoopyBeliefPropagation method*), 52  
`eraseAllEvidence()` (*pyAgrum.LoopyGibbsSampling method*), 82  
`eraseAllEvidence()` (*pyAgrum.LoopyImportanceSampling method*), 101  
`eraseAllEvidence()` (*pyAgrum.LoopyMonteCarloSampling method*), 89  
`eraseAllEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 95  
`eraseAllEvidence()` (*pyAgrum.MonteCarloSampling method*), 64  
`eraseAllEvidence()` (*pyAgrum.ShaferShenoyInference method*), 39  
`eraseAllEvidence()` (*pyAgrum.VariableElimination method*), 45  
`eraseAllEvidence()` (*pyAgrum.WeightedSampling method*), 70  
`eraseAllJointTargets()` (*pyAgrum.LazyPropagation method*), 32  
`eraseAllJointTargets()` (*pyAgrum.ShaferShenoyInference method*), 39  
`eraseAllMarginalTargets()` (*pyAgrum.LazyPropagation method*), 32  
`eraseAllMarginalTargets()` (*pyAgrum.ShaferShenoyInference method*), 39  
`eraseAllTargets()` (*pyAgrum.GibbsSampling method*), 58  
`eraseAllTargets()` (*pyAgrum.ImportanceSampling method*), 76  
`eraseAllTargets()` (*pyAgrum.LazyPropagation method*), 32  
`eraseAllTargets()` (*pyAgrum.LoopyBeliefPropagation method*), 52  
`eraseAllTargets()` (*pyAgrum.LoopyGibbsSampling method*), 82  
`eraseAllTargets()` (*pyAgrum.LoopyImportanceSampling method*), 101  
`eraseAllTargets()` (*pyAgrum.LoopyMonteCarloSampling method*), 89  
`eraseAllTargets()` (*pyAgrum.LoopyWeightedSampling method*), 95  
`eraseAllTargets()` (*pyAgrum.MonteCarloSampling method*), 64  
`eraseAllTargets()` (*pyAgrum.ShaferShenoyInference method*), 39  
`eraseAllTargets()` (*pyAgrum.VariableElimination method*), 45  
`eraseAllTargets()` (*pyAgrum.WeightedSampling method*), 70  
`eraseArc()` (*pyAgrum.BayesNet method*), 10  
`eraseArc()` (*pyAgrum.DAG method*), 118  
`eraseArc()` (*pyAgrum.DiGraph method*), 116  
`eraseArc()` (*pyAgrum.InfluenceDiagram method*), 210

- `eraseArc()` (*pyAgrum.MixedGraph* method), 127  
`eraseChildren()` (*pyAgrum.DAG* method), 118  
`eraseChildren()` (*pyAgrum.DiGraph* method), 116  
`eraseChildren()` (*pyAgrum.MixedGraph* method), 127  
`eraseEdge()` (*pyAgrum.CliqueGraph* method), 124  
`eraseEdge()` (*pyAgrum.MixedGraph* method), 128  
`eraseEdge()` (*pyAgrum.UndiGraph* method), 121  
`eraseEvidence()` (*pyAgrum.GibbsSampling* method), 58  
`eraseEvidence()` (*pyAgrum.ImportanceSampling* method), 76  
`eraseEvidence()` (*pyAgrum.InfluenceDiagramInference* method), 213  
`eraseEvidence()` (*pyAgrum.LazyPropagation* method), 32  
`eraseEvidence()` (*pyAgrum.LoopyBeliefPropagation* method), 52  
`eraseEvidence()` (*pyAgrum.LoopyGibbsSampling* method), 82  
`eraseEvidence()` (*pyAgrum.LoopyImportanceSampling* method), 101  
`eraseEvidence()` (*pyAgrum.LoopyMonteCarloSampling* method), 89  
`eraseEvidence()` (*pyAgrum.LoopyWeightedSampling* method), 95  
`eraseEvidence()` (*pyAgrum.MonteCarloSampling* method), 64  
`eraseEvidence()` (*pyAgrum.ShaferShenoyInference* method), 39  
`eraseEvidence()` (*pyAgrum.VariableElimination* method), 45  
`eraseEvidence()` (*pyAgrum.WeightedSampling* method), 70  
`eraseForbiddenArc()` (*pyAgrum.BN Learner* method), 107  
`eraseFromClique()` (*pyAgrum.CliqueGraph* method), 124  
`eraseJointTarget()` (*pyAgrum.LazyPropagation* method), 33  
`eraseJointTarget()` (*pyAgrum.ShaferShenoyInference* method), 39  
`eraseJointTarget()` (*pyAgrum.VariableElimination* method), 46  
`eraseLabels()` (*pyAgrum.LabelizedVariable* method), 134  
`eraseMandatoryArc()` (*pyAgrum.BN Learner* method), 107  
`eraseNeighbours()` (*pyAgrum.CliqueGraph* method), 124  
`eraseNeighbours()` (*pyAgrum.MixedGraph* method), 128  
`eraseNeighbours()` (*pyAgrum.UndiGraph* method), 121  
`eraseNode()` (*pyAgrum.CliqueGraph* method), 124  
`eraseNode()` (*pyAgrum.DAG* method), 118  
`eraseNode()` (*pyAgrum.DiGraph* method), 116  
`eraseNode()` (*pyAgrum.MixedGraph* method), 128  
`eraseNode()` (*pyAgrum.UndiGraph* method), 121  
`eraseParents()` (*pyAgrum.DAG* method), 118  
`eraseParents()` (*pyAgrum.DiGraph* method), 116  
`eraseParents()` (*pyAgrum.MixedGraph* method), 128  
`erasePossibleEdge()` (*pyAgrum.BN Learner* method), 107  
`eraseTarget()` (*pyAgrum.GibbsSampling* method), 58  
`eraseTarget()` (*pyAgrum.ImportanceSampling* method), 76  
`eraseTarget()` (*pyAgrum.LazyPropagation* method), 33  
`eraseTarget()` (*pyAgrum.LoopyBeliefPropagation* method), 52  
`eraseTarget()` (*pyAgrum.LoopyGibbsSampling* method), 83  
`eraseTarget()` (*pyAgrum.LoopyImportanceSampling* method), 101  
`eraseTarget()` (*pyAgrum.LoopyMonteCarloSampling* method), 89  
`eraseTarget()` (*pyAgrum.LoopyWeightedSampling* method), 95  
`eraseTarget()` (*pyAgrum.MonteCarloSampling* method), 64  
`eraseTarget()` (*pyAgrum.ShaferShenoyInference* method), 40  
`eraseTarget()` (*pyAgrum.VariableElimination* method), 46  
`eraseTarget()` (*pyAgrum.WeightedSampling* method), 70  
`eraseTicks()` (*pyAgrum.DiscretizedVariable* method), 137  
`errorCallStack()` (*pyAgrum.DefaultInLabel* method), 221  
`errorCallStack()` (*pyAgrum.DuplicateElement* method), 221  
`errorCallStack()` (*pyAgrum.DuplicateLabel* method), 222  
`errorCallStack()` (*pyAgrum.EmptyBSTree* method), 222  
`errorCallStack()` (*pyAgrum.EmptySet* method), 222  
`errorCallStack()` (*pyAgrum.FatalError* method), 223  
`errorCallStack()` (*pyAgrum.FormatNotFound* method), 223

`errorCallStack()` (`pyAgrum.GraphError` `method`), 224

`errorCallStack()` (`pyAgrum.GumException` `method`), 223

`errorCallStack()` (`pyAgrum.IdError` `method`), 224

`errorCallStack()` (`pyAgrum.InvalidArc` `method`), 225

`errorCallStack()` (`pyAgrum.InvalidArgument` `method`), 225

`errorCallStack()` (`pyA-grum.InvalidArgumentsNumber` `method`), 225

`errorCallStack()` (`pyA-grum.InvalidDirectedCycle` `method`), 226

`errorCallStack()` (`pyAgrum.InvalidEdge` `method`), 226

`errorCallStack()` (`pyAgrum.InvalidNode` `method`), 226

`errorCallStack()` (`pyAgrum.IOError` `method`), 224

`errorCallStack()` (`pyAgrum.NoChild` `method`), 227

`errorCallStack()` (`pyAgrum.NoNeighbour` `method`), 227

`errorCallStack()` (`pyAgrum.NoParent` `method`), 227

`errorCallStack()` (`pyAgrum.NotFound` `method`), 228

`errorCallStack()` (`pyAgrum.NullElement` `method`), 228

`errorCallStack()` (`pyA-grum.OperationNotAllowed` `method`), 228

`errorCallStack()` (`pyAgrum.OutOfBounds` `method`), 229

`errorCallStack()` (`pyAgrum.OutOfLowerBound` `method`), 229

`errorCallStack()` (`pyAgrum.OutOfUpperBound` `method`), 229

`errorCallStack()` (`pyAgrum.ReferenceError` `method`), 230

`errorCallStack()` (`pyAgrum.SizeError` `method`), 230

`errorCallStack()` (`pyAgrum.SyntaxError` `method`), 231

`errorCallStack()` (`pyAgrum.UndefinedElement` `method`), 231

`errorCallStack()` (`pyA-grum.UndefinedIteratorKey` `method`), 231

`errorCallStack()` (`pyA-grum.UndefinedIteratorValue` `method`), 232

`errorCallStack()` (`pyA-grum.UnknownLabelInDatabase` `method`), 232

`errorContent()` (`pyAgrum.DefaultInLabel` `method`), 221

`errorContent()` (`pyAgrum.DuplicateElement` `method`), 221

`errorContent()` (`pyAgrum.DuplicateLabel` `method`), 222

`errorContent()` (`pyAgrum.EmptyBSTree` `method`), 222

`errorContent()` (`pyAgrum.EmptySet` `method`), 222

`errorContent()` (`pyAgrum.FatalError` `method`), 223

`errorContent()` (`pyAgrum.FormatNotFound` `method`), 223

`errorContent()` (`pyAgrum.GraphError` `method`), 224

`errorContent()` (`pyAgrum.GumException` `method`), 223

`errorContent()` (`pyAgrum.IdError` `method`), 224

`errorContent()` (`pyAgrum.InvalidArc` `method`), 225

`errorContent()` (`pyAgrum.InvalidArgument` `method`), 225

`errorContent()` (`pyA-grum.InvalidArgumentsNumber` `method`), 225

`errorContent()` (`pyAgrum.InvalidDirectedCycle` `method`), 226

`errorContent()` (`pyAgrum.InvalidEdge` `method`), 226

`errorContent()` (`pyAgrum.InvalidNode` `method`), 226

`errorContent()` (`pyAgrum.IOError` `method`), 224

`errorContent()` (`pyAgrum.NoChild` `method`), 227

`errorContent()` (`pyAgrum.NoNeighbour` `method`), 227

`errorContent()` (`pyAgrum.NoParent` `method`), 228

`errorContent()` (`pyAgrum.NotFound` `method`), 228

`errorContent()` (`pyAgrum.NullElement` `method`), 228

`errorContent()` (`pyAgrum.OperationNotAllowed` `method`), 229

`errorContent()` (`pyAgrum.OutOfBounds` `method`), 229

`errorContent()` (`pyAgrum.OutOfLowerBound` `method`), 229

`errorContent()` (`pyAgrum.OutOfUpperBound` `method`), 230

`errorContent()` (`pyAgrum.ReferenceError` `method`), 230

`errorContent()` (`pyAgrum.SizeError` `method`), 230

`errorContent()` (`pyAgrum.SyntaxError` `method`), 231

`errorContent()` (`pyAgrum.UndefinedElement` `method`), 231

`errorContent()` (`pyAgrum.UndefinedIteratorKey` `method`), 231

`errorContent()` (`pyA-grum.UndefinedIteratorValue` `method`), 232

`errorContent()` (`pyA-`



`grum.UnknownLabelInDatabase` method), 232  
`errorType()` (`pyAgrum.DefaultInLabel` method), 221  
`errorType()` (`pyAgrum.DuplicateElement` method), 221  
`errorType()` (`pyAgrum.DuplicateLabel` method), 222  
`errorType()` (`pyAgrum.EmptyBSTree` method), 222  
`errorType()` (`pyAgrum.EmptySet` method), 222  
`errorType()` (`pyAgrum.FatalError` method), 223  
`errorType()` (`pyAgrum.FormatNotFound` method), 223  
`errorType()` (`pyAgrum.GraphError` method), 224  
`errorType()` (`pyAgrum.GumException` method), 223  
`errorType()` (`pyAgrum.IdError` method), 225  
`errorType()` (`pyAgrum.InvalidArc` method), 225  
`errorType()` (`pyAgrum.InvalidArgument` method), 225  
`errorType()` (`pyAgrum.InvalidArgumentsNumber` method), 226  
`errorType()` (`pyAgrum.InvalidDirectedCycle` method), 226  
`errorType()` (`pyAgrum.InvalidEdge` method), 226  
`errorType()` (`pyAgrum.InvalidNode` method), 227  
`errorType()` (`pyAgrum.IOError` method), 224  
`errorType()` (`pyAgrum.NoChild` method), 227  
`errorType()` (`pyAgrum.NoNeighbour` method), 227  
`errorType()` (`pyAgrum.NoParent` method), 228  
`errorType()` (`pyAgrum.NotFound` method), 228  
`errorType()` (`pyAgrum.NullElement` method), 228  
`errorType()` (`pyAgrum.OperationNotAllowed` method), 229  
`errorType()` (`pyAgrum.OutOfBounds` method), 229  
`errorType()` (`pyAgrum.OutOfLowerBound` method), 229  
`errorType()` (`pyAgrum.OutOfUpperBound` method), 230  
`errorType()` (`pyAgrum.ReferenceError` method), 230  
`errorType()` (`pyAgrum.SizeError` method), 230  
`errorType()` (`pyAgrum.SyntaxError` method), 231  
`errorType()` (`pyAgrum.UndefinedElement` method), 231  
`errorType()` (`pyAgrum.UndefinedIteratorKey` method), 232  
`errorType()` (`pyAgrum.UndefinedIteratorValue` method), 232  
`errorType()` (`pyAgrum.UnknownLabelInDatabase` method), 232  
`EssentialGraph` (class in `pyAgrum`), 22  
`eval()` (`pyAgrum.causal.ASTsum` method), 185  
`eval()` (`pyAgrum.causal.CausalFormula` method), 181  
`evidenceImpact()` (`pyAgrum.GibbsSampling` method), 58  
`evidenceImpact()` (`pyAgrum.ImportanceSampling` method), 77  
`evidenceImpact()` (`pyAgrum.LazyPropagation` method), 33  
`evidenceImpact()` (`pyAgrum.LoopyBeliefPropagation` method), 52  
`evidenceImpact()` (`pyAgrum.LoopyGibbsSampling` method), 83  
`evidenceImpact()` (`pyAgrum.LoopyImportanceSampling` method), 102  
`evidenceImpact()` (`pyAgrum.LoopyMonteCarloSampling` method), 89  
`evidenceImpact()` (`pyAgrum.LoopyWeightedSampling` method), 95  
`evidenceImpact()` (`pyAgrum.MonteCarloSampling` method), 65  
`evidenceImpact()` (`pyAgrum.ShaferShenoyInference` method), 40  
`evidenceImpact()` (`pyAgrum.VariableElimination` method), 46  
`evidenceImpact()` (`pyAgrum.WeightedSampling` method), 71  
`evidenceJointImpact()` (`pyAgrum.LazyPropagation` method), 33  
`evidenceJointImpact()` (`pyAgrum.ShaferShenoyInference` method), 40  
`evidenceJointImpact()` (`pyAgrum.VariableElimination` method), 46  
`evidenceProbability()` (`pyAgrum.LazyPropagation` method), 33  
`evidenceProbability()` (`pyAgrum.ShaferShenoyInference` method), 40  
`ExactBNdistance` (class in `pyAgrum`), 17  
`existsArc()` (`pyAgrum.DAG` method), 119  
`existsArc()` (`pyAgrum.DiGraph` method), 116  
`existsArc()` (`pyAgrum.MixedGraph` method), 128  
`existsEdge()` (`pyAgrum.CliqueGraph` method), 124  
`existsEdge()` (`pyAgrum.MixedGraph` method), 128  
`existsEdge()` (`pyAgrum.UndiGraph` method), 121  
`existsNode()` (`pyAgrum.CliqueGraph` method), 124  
`existsNode()` (`pyAgrum.DAG` method), 119  
`existsNode()` (`pyAgrum.DiGraph` method), 116  
`existsNode()` (`pyAgrum.MixedGraph` method), 128  
`existsNode()` (`pyAgrum.UndiGraph` method), 121  
`existsPathBetween()` (`pyAgrum.InfluenceDiagram` method), 210  
`extract()` (`pyAgrum.Potential` method), 150

## F

fastBN() (in module *pyAgrum*), 215  
 fastPrototype() (*pyAgrum.BayesNet* static method), 10  
 FatalError, 223  
 fill() (*pyAgrum.Potential* method), 150  
 fillConstraint() (*pyAgrum.CredalNet* method), 197  
 fillConstraints() (*pyAgrum.CredalNet* method), 197  
 fillWith() (*pyAgrum.Potential* method), 150  
 fillWithFunction() (*pyAgrum.Potential* method), 150  
 findAll() (*pyAgrum.Potential* method), 151  
 first() (*pyAgrum.Arc* method), 113  
 first() (*pyAgrum.Edge* method), 114  
 forDarkTheme() (in module *pyAgrum.lib.bn2graph*), 170  
 forLightTheme() (in module *pyAgrum.lib.bn2graph*), 170  
 FormatNotFound, 223  
 fromdict() (*pyAgrum.Instantiation* method), 146

## G

G2() (*pyAgrum.BN Learner* method), 106  
 generateCPT() (*pyAgrum.BayesNet* method), 11  
 generateCPTs() (*pyAgrum.BayesNet* method), 11  
 get() (*pyAgrum.Potential* method), 151  
 get\_CPT\_max() (*pyAgrum.CredalNet* method), 198  
 get\_CPT\_min() (*pyAgrum.CredalNet* method), 198  
 getAlltheSystems() (*pyAgrum.PRMexplorer* method), 191  
 getBestDecisionChoice() (*pyAgrum.InfluenceDiagramInference* method), 213  
 getBlackInTheme() (in module *pyAgrum.lib.bn2graph*), 170  
 getBN() (in module *pyAgrum.lib.notebook*), 157, 164  
 getBNDiff() (in module *pyAgrum.lib.notebook*), 157  
 getCausalImpact() (in module *pyAgrum.causal.notebook*), 186  
 getCausalModel() (in module *pyAgrum.causal.notebook*), 187  
 getDecisionGraph() (*pyAgrum.InfluenceDiagram* method), 210  
 getDecisionOrder() (*pyAgrum.InfluenceDiagram* method), 210  
 getDirectSubClass() (*pyAgrum.PRMexplorer* method), 190  
 getDirectSubInterfaces() (*pyAgrum.PRMexplorer* method), 190  
 getDirectSubTypes() (*pyAgrum.PRMexplorer* method), 191  
 getDot() (in module *pyAgrum.lib.notebook*), 158, 164  
 getGraph() (in module *pyAgrum.lib.notebook*), 158, 164

getImplementations() (*pyAgrum.PRMexplorer* method), 191  
 getInference() (in module *pyAgrum.lib.notebook*), 158, 165  
 getInferenceEngine() (in module *pyAgrum.lib.notebook*), 158  
 getInfluenceDiagram() (in module *pyAgrum.lib.notebook*), 159, 167  
 getInformation() (in module *pyAgrum.lib.notebook*), 159, 167  
 getJunctionTree() (in module *pyAgrum.lib.notebook*), 159, 166  
 getLabelMap() (*pyAgrum.PRMexplorer* method), 191  
 getLabels() (*pyAgrum.PRMexplorer* method), 191  
 getMaxNumberOfThreads() (in module *pyAgrum*), 219  
 getMEU() (*pyAgrum.InfluenceDiagramInference* method), 213  
 getNumberOfLogicalProcessors() (in module *pyAgrum*), 219  
 getPosterior() (in module *pyAgrum*), 215  
 getPosterior() (in module *pyAgrum.lib.notebook*), 159, 163  
 getPotential() (in module *pyAgrum.lib.notebook*), 159, 163  
 getSideBySide() (in module *pyAgrum.lib.notebook*), 159  
 getSuperClass() (*pyAgrum.PRMexplorer* method), 191  
 getSuperInterface() (*pyAgrum.PRMexplorer* method), 191  
 getSuperType() (*pyAgrum.PRMexplorer* method), 191  
 GibbsBNdistance (class in *pyAgrum*), 17  
 GibbsSampling (class in *pyAgrum*), 56  
 GraphError, 224  
 GumException, 223

## H

H() (*pyAgrum.GibbsSampling* method), 56  
 H() (*pyAgrum.ImportanceSampling* method), 74  
 H() (*pyAgrum.LazyPropagation* method), 30  
 H() (*pyAgrum.LoopyBeliefPropagation* method), 50  
 H() (*pyAgrum.LoopyGibbsSampling* method), 81  
 H() (*pyAgrum.LoopyImportanceSampling* method), 99  
 H() (*pyAgrum.LoopyMonteCarloSampling* method), 87  
 H() (*pyAgrum.LoopyWeightedSampling* method), 93  
 H() (*pyAgrum.MonteCarloSampling* method), 62  
 H() (*pyAgrum.ShaferShenoyInference* method), 37  
 H() (*pyAgrum.VariableElimination* method), 44  
 H() (*pyAgrum.WeightedSampling* method), 68  
 hamming() (*pyAgrum.Instantiation* method), 146  
 hardEvidenceNodes() (*pyAgrum.GibbsSampling* method), 58  
 hardEvidenceNodes() (*pyAgrum.ImportanceSampling* method), 77

---

`hardEvidenceNodes()` (`pyAgrum.LazyPropagation` method), 34  
`hardEvidenceNodes()` (`pyAgrum.LoopyBeliefPropagation` method), 52  
`hardEvidenceNodes()` (`pyAgrum.LoopyGibbsSampling` method), 83  
`hardEvidenceNodes()` (`pyAgrum.LoopyImportanceSampling` method), 102  
`hardEvidenceNodes()` (`pyAgrum.LoopyMonteCarloSampling` method), 90  
`hardEvidenceNodes()` (`pyAgrum.LoopyWeightedSampling` method), 96  
`hardEvidenceNodes()` (`pyAgrum.MonteCarloSampling` method), 65  
`hardEvidenceNodes()` (`pyAgrum.ShaferShenoyInference` method), 40  
`hardEvidenceNodes()` (`pyAgrum.VariableElimination` method), 46  
`hardEvidenceNodes()` (`pyAgrum.WeightedSampling` method), 71  
`hasComputedCPTMinMax()` (`pyAgrum.CredalNet` method), 198  
`hasDirectedPath()` (`pyAgrum.DAG` method), 119  
`hasDirectedPath()` (`pyAgrum.DiGraph` method), 116  
`hasDirectedPath()` (`pyAgrum.MixedGraph` method), 128  
`hasEvidence()` (`pyAgrum.GibbsSampling` method), 59  
`hasEvidence()` (`pyAgrum.ImportanceSampling` method), 77  
`hasEvidence()` (`pyAgrum.LazyPropagation` method), 34  
`hasEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 52  
`hasEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 83  
`hasEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 102  
`hasEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 90  
`hasEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 96  
`hasEvidence()` (`pyAgrum.MonteCarloSampling` method), 65  
`hasEvidence()` (`pyAgrum.ShaferShenoyInference` method), 40  
`hasEvidence()` (`pyAgrum.VariableElimination` method), 47  
`hasEvidence()` (`pyAgrum.WeightedSampling` method), 71  
`hasHardEvidence()` (`pyAgrum.GibbsSampling` method), 59  
`hasHardEvidence()` (`pyAgrum.ImportanceSampling` method), 77  
`hasHardEvidence()` (`pyAgrum.LazyPropagation` method), 34  
`hasHardEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 53  
`hasHardEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 83  
`hasHardEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 102  
`hasHardEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method), 90  
`hasHardEvidence()` (`pyAgrum.LoopyWeightedSampling` method), 96  
`hasHardEvidence()` (`pyAgrum.MonteCarloSampling` method), 65  
`hasHardEvidence()` (`pyAgrum.ShaferShenoyInference` method), 41  
`hasHardEvidence()` (`pyAgrum.VariableElimination` method), 47  
`hasHardEvidence()` (`pyAgrum.WeightedSampling` method), 71  
`hasMissingValues()` (`pyAgrum.BN Learner` method), 108  
`hasRunningIntersection()` (`pyAgrum.CliqueGraph` method), 124  
`hasSameStructure()` (`pyAgrum.BayesNet` method), 11  
`hasSameStructure()` (`pyAgrum.BayesNetFragment` method), 26  
`hasSameStructure()` (`pyAgrum.InfluenceDiagram` method), 210  
`hasSameStructure()` (`pyAgrum.MarkovBlanket` method), 24  
`hasSoftEvidence()` (`pyAgrum.GibbsSampling` method), 59  
`hasSoftEvidence()` (`pyAgrum.ImportanceSampling` method), 77  
`hasSoftEvidence()` (`pyAgrum.LazyPropagation` method), 34  
`hasSoftEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 53  
`hasSoftEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 84  
`hasSoftEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 102  
`hasSoftEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method),

- 90
- `hasSoftEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 96
- `hasSoftEvidence()` (*pyAgrum.MonteCarloSampling method*), 65
- `hasSoftEvidence()` (*pyAgrum.ShaferShenoyInference method*), 41
- `hasSoftEvidence()` (*pyAgrum.VariableElimination method*), 47
- `hasSoftEvidence()` (*pyAgrum.WeightedSampling method*), 71
- `hasUndirectedCycle()` (*pyAgrum.CliqueGraph method*), 124
- `hasUndirectedCycle()` (*pyAgrum.MixedGraph method*), 128
- `hasUndirectedCycle()` (*pyAgrum.UndiGraph method*), 121
- `head()` (*pyAgrum.Arc method*), 113
- `HedgeException` (*class in pyAgrum.causal*), 186
- `history()` (*pyAgrum.BN Learner method*), 108
- `history()` (*pyAgrum.CN LoopyPropagation method*), 203
- `history()` (*pyAgrum.CN MonteCarloSampling method*), 201
- `history()` (*pyAgrum.Gibbs BN distance method*), 18
- `history()` (*pyAgrum.Gibbs Sampling method*), 59
- `history()` (*pyAgrum.ImportanceSampling method*), 78
- `history()` (*pyAgrum.Loopy BeliefPropagation method*), 53
- `history()` (*pyAgrum.Loopy Gibbs Sampling method*), 84
- `history()` (*pyAgrum.Loopy ImportanceSampling method*), 103
- `history()` (*pyAgrum.Loopy MonteCarloSampling method*), 90
- `history()` (*pyAgrum.Loopy WeightedSampling method*), 96
- `history()` (*pyAgrum.MonteCarloSampling method*), 65
- `history()` (*pyAgrum.WeightedSampling method*), 72
- I**
- `I()` (*pyAgrum.LazyPropagation method*), 31
- `I()` (*pyAgrum.ShaferShenoyInference method*), 37
- `identifyingIntervention()` (*in module pyAgrum.causal*), 182
- `IdError`, 224
- `idFromName()` (*pyAgrum.BayesNet method*), 11
- `idFromName()` (*pyAgrum.BayesNetFragment method*), 26
- `idFromName()` (*pyAgrum.BN Learner method*), 108
- `idFromName()` (*pyAgrum.causal.CausalModel method*), 180
- `idFromName()` (*pyAgrum.InfluenceDiagram method*), 210
- `idmLearning()` (*pyAgrum.CredalNet method*), 198
- `ids()` (*pyAgrum.BayesNet method*), 11
- `ids()` (*pyAgrum.CliqueGraph method*), 125
- `ids()` (*pyAgrum.DAG method*), 119
- `ids()` (*pyAgrum.DiGraph method*), 116
- `ids()` (*pyAgrum.EssentialGraph method*), 22
- `ids()` (*pyAgrum.InfluenceDiagram method*), 211
- `ids()` (*pyAgrum.MixedGraph method*), 129
- `ids()` (*pyAgrum.UndiGraph method*), 121
- `ImportanceSampling` (*class in pyAgrum*), 74
- `inc()` (*pyAgrum.Instantiation method*), 146
- `incIn()` (*pyAgrum.Instantiation method*), 146
- `incNotVar()` (*pyAgrum.Instantiation method*), 146
- `incOut()` (*pyAgrum.Instantiation method*), 146
- `incVar()` (*pyAgrum.Instantiation method*), 146
- `index()` (*pyAgrum.DiscreteVariable method*), 132
- `index()` (*pyAgrum.DiscretizedVariable method*), 137
- `index()` (*pyAgrum.LabelizedVariable method*), 134
- `index()` (*pyAgrum.RangeVariable method*), 139
- `inferenceType()` (*pyAgrum.CN LoopyPropagation method*), 203
- `InfluenceDiagram` (*class in pyAgrum*), 207
- `influenceDiagram()` (*pyAgrum.InfluenceDiagramInference method*), 214
- `InfluenceDiagramInference` (*class in pyAgrum*), 213
- `initApproximationScheme()` (*pyAgrum.Gibbs BN distance method*), 18
- `initRandom()` (*in module pyAgrum*), 218
- `inOverflow()` (*pyAgrum.Instantiation method*), 146
- `insertEvidence()` (*pyAgrum.InfluenceDiagramInference method*), 214
- `insertEvidenceFile()` (*pyAgrum.CN LoopyPropagation method*), 203
- `insertEvidenceFile()` (*pyAgrum.CN MonteCarloSampling method*), 201
- `insertModalsFile()` (*pyAgrum.CN LoopyPropagation method*), 203
- `insertModalsFile()` (*pyAgrum.CN MonteCarloSampling method*), 201
- `installAscendants()` (*pyAgrum.BayesNetFragment method*), 26
- `installCPT()` (*pyAgrum.BayesNetFragment method*), 26
- `installMarginal()` (*pyAgrum.BayesNetFragment method*), 27
- `installNode()` (*pyAgrum.BayesNetFragment method*), 27



- Instantiation (*class in pyAgrum*), 144
- instantiation() (*pyAgrum.CredalNet method*), 198
- interAttributes() (*pyAgrum.PRMexplorer method*), 192
- interfaces() (*pyAgrum.PRMexplorer method*), 192
- interReferences() (*pyAgrum.PRMexplorer method*), 192
- intervalToCredal() (*pyAgrum.CredalNet method*), 198
- intervalToCredalWithFiles() (*pyAgrum.CredalNet method*), 199
- InvalidArc, 225
- InvalidArgument, 225
- InvalidArgumentsNumber, 225
- InvalidDirectedCycle, 226
- InvalidEdge, 226
- InvalidNode, 226
- IOError, 224
- isAttribute() (*pyAgrum.PRMexplorer method*), 192
- isChanceNode() (*pyAgrum.InfluenceDiagram method*), 211
- isClass() (*pyAgrum.PRMexplorer method*), 192
- isDecisionNode() (*pyAgrum.InfluenceDiagram method*), 211
- isDrawnAtRandom() (*pyAgrum.GibbsBNdistance method*), 18
- isDrawnAtRandom() (*pyAgrum.GibbsSampling method*), 59
- isDrawnAtRandom() (*pyAgrum.LoopyGibbsSampling method*), 84
- isEnabledEpsilon() (*pyAgrum.GibbsBNdistance method*), 19
- isEnabledMaxIter() (*pyAgrum.GibbsBNdistance method*), 19
- isEnabledMaxTime() (*pyAgrum.GibbsBNdistance method*), 19
- isEnabledMinEpsilonRate() (*pyAgrum.GibbsBNdistance method*), 19
- isInstalledNode() (*pyAgrum.BayesNetFragment method*), 27
- isInterface() (*pyAgrum.PRMexplorer method*), 192
- isJoinTree() (*pyAgrum.CliqueGraph method*), 125
- isJointTarget() (*pyAgrum.LazyPropagation method*), 34
- isJointTarget() (*pyAgrum.ShaferShenoyInference method*), 41
- isJointTarget() (*pyAgrum.VariableElimination method*), 47
- isLabel() (*pyAgrum.LabelizedVariable method*), 134
- isNonZeroMap() (*pyAgrum.Potential method*), 151
- isOMP() (*in module pyAgrum*), 219
- isSeparatelySpecified() (*pyAgrum.CredalNet method*), 199
- isTarget() (*pyAgrum.GibbsSampling method*), 59
- isTarget() (*pyAgrum.ImportanceSampling method*), 78
- isTarget() (*pyAgrum.LazyPropagation method*), 34
- isTarget() (*pyAgrum.LoopyBeliefPropagation method*), 53
- isTarget() (*pyAgrum.LoopyGibbsSampling method*), 84
- isTarget() (*pyAgrum.LoopyImportanceSampling method*), 103
- isTarget() (*pyAgrum.LoopyMonteCarloSampling method*), 90
- isTarget() (*pyAgrum.LoopyWeightedSampling method*), 97
- isTarget() (*pyAgrum.MonteCarloSampling method*), 66
- isTarget() (*pyAgrum.ShaferShenoyInference method*), 41
- isTarget() (*pyAgrum.VariableElimination method*), 47
- isTarget() (*pyAgrum.WeightedSampling method*), 72
- isTick() (*pyAgrum.DiscretizedVariable method*), 137
- isType() (*pyAgrum.PRMexplorer method*), 192
- isUtilityNode() (*pyAgrum.InfluenceDiagram method*), 211
- ## J
- jointMutualInformation() (*pyAgrum.LazyPropagation method*), 35
- jointMutualInformation() (*pyAgrum.ShaferShenoyInference method*), 42
- jointMutualInformation() (*pyAgrum.VariableElimination method*), 48
- jointPosterior() (*pyAgrum.LazyPropagation method*), 35
- jointPosterior() (*pyAgrum.ShaferShenoyInference method*), 42
- jointPosterior() (*pyAgrum.VariableElimination method*), 48
- jointProbability() (*pyAgrum.BayesNet method*), 11
- jointProbability() (*pyAgrum.BayesNetFragment method*), 27
- joinTree() (*pyAgrum.LazyPropagation method*), 35
- joinTree() (*pyAgrum.ShaferShenoyInference method*), 41
- jointTargets() (*pyAgrum.LazyPropagation method*), 35
- jointTargets() (*pyAgrum.ShaferShenoyInference method*), 42

- jointTargets() (*pyAgrum.VariableElimination method*), 48
- junctionTree() (*pyAgrum.JunctionTreeGenerator method*), 21
- junctionTree() (*pyAgrum.LazyPropagation method*), 35
- junctionTree() (*pyAgrum.ShaferShenoyInference method*), 42
- junctionTree() (*pyAgrum.VariableElimination method*), 48
- JunctionTreeGenerator (*class in pyAgrum*), 20
- junctionTreeToDot() (*pyAgrum.InfluenceDiagramInference method*), 214
- ## K
- KL() (*pyAgrum.Potential method*), 149
- knw (*pyAgrum.causal.ASTposteriorProba attribute*), 186
- ## L
- label() (*pyAgrum.DiscreteVariable method*), 132
- label() (*pyAgrum.DiscretizedVariable method*), 137
- label() (*pyAgrum.LabelizedVariable method*), 134
- label() (*pyAgrum.RangeVariable method*), 139
- LabelizedVariable (*class in pyAgrum*), 133
- labels() (*pyAgrum.DiscreteVariable method*), 132
- labels() (*pyAgrum.DiscretizedVariable method*), 137
- labels() (*pyAgrum.LabelizedVariable method*), 135
- labels() (*pyAgrum.RangeVariable method*), 140
- lagrangeNormalization() (*pyAgrum.CredalNet method*), 199
- latentVariables() (*pyAgrum.BN Learner method*), 108
- latentVariablesIds() (*pyAgrum.causal.CausalModel method*), 180
- latexQuery() (*pyAgrum.causal.CausalFormula method*), 181
- LazyPropagation (*class in pyAgrum*), 30
- learnBN() (*pyAgrum.BN Learner method*), 108
- learnDAG() (*pyAgrum.BN Learner method*), 108
- learnMixedStructure() (*pyAgrum.BN Learner method*), 108
- learnParameters() (*pyAgrum.BN Learner method*), 108
- line() (*pyAgrum.SyntaxError method*), 231
- load() (*pyAgrum.PRMexplorer method*), 192
- loadBIF() (*pyAgrum.BayesNet method*), 11
- loadBIFXML() (*pyAgrum.BayesNet method*), 11
- loadBIFXML() (*pyAgrum.InfluenceDiagram method*), 211
- loadBN() (*in module pyAgrum*), 216
- loadDSL() (*pyAgrum.BayesNet method*), 12
- loadID() (*in module pyAgrum*), 216
- loadNET() (*pyAgrum.BayesNet method*), 12
- loadO3PRM() (*pyAgrum.BayesNet method*), 12
- loadUAI() (*pyAgrum.BayesNet method*), 12
- log10DomainSize() (*pyAgrum.BayesNet method*), 13
- log10DomainSize() (*pyAgrum.BayesNetFragment method*), 27
- log10DomainSize() (*pyAgrum.InfluenceDiagram method*), 211
- log2JointProbability() (*pyAgrum.BayesNet method*), 13
- log2JointProbability() (*pyAgrum.BayesNetFragment method*), 27
- log2likelihood() (*pyAgrum.BNDatabaseGenerator method*), 16
- logLikelihood() (*pyAgrum.BN Learner method*), 109
- loopIn() (*pyAgrum.Potential method*), 151
- LoopyBeliefPropagation (*class in pyAgrum*), 50
- LoopyGibbsSampling (*class in pyAgrum*), 80
- LoopyImportanceSampling (*class in pyAgrum*), 99
- LoopyMonteCarloSampling (*class in pyAgrum*), 87
- LoopyWeightedSampling (*class in pyAgrum*), 93
- ## M
- makeInference() (*pyAgrum.CNLoopyPropagation method*), 203
- makeInference() (*pyAgrum.CNMonteCarloSampling method*), 201
- makeInference() (*pyAgrum.GibbsSampling method*), 60
- makeInference() (*pyAgrum.ImportanceSampling method*), 78
- makeInference() (*pyAgrum.InfluenceDiagramInference method*), 214
- makeInference() (*pyAgrum.LazyPropagation method*), 35
- makeInference() (*pyAgrum.LoopyBeliefPropagation method*), 53
- makeInference() (*pyAgrum.LoopyGibbsSampling method*), 84
- makeInference() (*pyAgrum.LoopyImportanceSampling method*), 103
- makeInference() (*pyAgrum.LoopyMonteCarloSampling method*), 91
- makeInference() (*pyAgrum.LoopyWeightedSampling method*), 97
- makeInference() (*pyAgrum.MonteCarloSampling method*), 66

`makeInference()` (`pyAgrum.ShaferShenoyInference` method), 42  
`makeInference()` (`pyAgrum.VariableElimination` method), 48  
`makeInference()` (`pyAgrum.WeightedSampling` method), 72  
`marginalMax()` (`pyAgrum.CNLoopyPropagation` method), 203  
`marginalMax()` (`pyAgrum.CNMonteCarloSampling` method), 201  
`marginalMin()` (`pyAgrum.CNLoopyPropagation` method), 204  
`marginalMin()` (`pyAgrum.CNMonteCarloSampling` method), 201  
`margMaxIn()` (`pyAgrum.Potential` method), 151  
`margMaxOut()` (`pyAgrum.Potential` method), 151  
`margMinIn()` (`pyAgrum.Potential` method), 151  
`margMinOut()` (`pyAgrum.Potential` method), 152  
`margProdIn()` (`pyAgrum.Potential` method), 152  
`margProdOut()` (`pyAgrum.Potential` method), 152  
`margSumIn()` (`pyAgrum.Potential` method), 152  
`margSumOut()` (`pyAgrum.Potential` method), 152  
`MarkovBlanket` (class in `pyAgrum`), 23  
`max()` (`pyAgrum.Potential` method), 152  
`maxIter()` (`pyAgrum.BN Learner` method), 109  
`maxIter()` (`pyAgrum.CNLoopyPropagation` method), 204  
`maxIter()` (`pyAgrum.CNMonteCarloSampling` method), 201  
`maxIter()` (`pyAgrum.GibbsBNdistance` method), 19  
`maxIter()` (`pyAgrum.GibbsSampling` method), 60  
`maxIter()` (`pyAgrum.ImportanceSampling` method), 78  
`maxIter()` (`pyAgrum.LoopyBeliefPropagation` method), 53  
`maxIter()` (`pyAgrum.LoopyGibbsSampling` method), 84  
`maxIter()` (`pyAgrum.LoopyImportanceSampling` method), 103  
`maxIter()` (`pyAgrum.LoopyMonteCarloSampling` method), 91  
`maxIter()` (`pyAgrum.LoopyWeightedSampling` method), 97  
`maxIter()` (`pyAgrum.MonteCarloSampling` method), 66  
`maxIter()` (`pyAgrum.WeightedSampling` method), 72  
`maxNonOne()` (`pyAgrum.Potential` method), 152  
`maxNonOneParam()` (`pyAgrum.BayesNet` method), 13  
`maxNonOneParam()` (`pyAgrum.BayesNetFragment` method), 27  
`maxParam()` (`pyAgrum.BayesNet` method), 13  
`maxParam()` (`pyAgrum.BayesNetFragment` method), 27  
`maxTime()` (`pyAgrum.BN Learner` method), 109  
`maxTime()` (`pyAgrum.CNLoopyPropagation` method), 204  
`maxTime()` (`pyAgrum.CNMonteCarloSampling` method), 201  
`maxTime()` (`pyAgrum.GibbsBNdistance` method), 19  
`maxTime()` (`pyAgrum.GibbsSampling` method), 60  
`maxTime()` (`pyAgrum.ImportanceSampling` method), 78  
`maxTime()` (`pyAgrum.LoopyBeliefPropagation` method), 54  
`maxTime()` (`pyAgrum.LoopyGibbsSampling` method), 84  
`maxTime()` (`pyAgrum.LoopyImportanceSampling` method), 103  
`maxTime()` (`pyAgrum.LoopyMonteCarloSampling` method), 91  
`maxTime()` (`pyAgrum.LoopyWeightedSampling` method), 97  
`maxTime()` (`pyAgrum.MonteCarloSampling` method), 66  
`maxTime()` (`pyAgrum.WeightedSampling` method), 72  
`maxVal()` (`pyAgrum.RangeVariable` method), 140  
`maxVarDomainSize()` (`pyAgrum.BayesNet` method), 13  
`maxVarDomainSize()` (`pyAgrum.BayesNetFragment` method), 28  
`messageApproximationScheme()` (`pyAgrum.BN Learner` method), 109  
`messageApproximationScheme()` (`pyAgrum.CNLoopyPropagation` method), 204  
`messageApproximationScheme()` (`pyAgrum.CNMonteCarloSampling` method), 201  
`messageApproximationScheme()` (`pyAgrum.GibbsBNdistance` method), 19  
`messageApproximationScheme()` (`pyAgrum.GibbsSampling` method), 60  
`messageApproximationScheme()` (`pyAgrum.ImportanceSampling` method), 78  
`messageApproximationScheme()` (`pyAgrum.LoopyBeliefPropagation` method), 54  
`messageApproximationScheme()` (`pyAgrum.LoopyGibbsSampling` method), 85  
`messageApproximationScheme()` (`pyAgrum.LoopyImportanceSampling` method), 103  
`messageApproximationScheme()` (`pyAgrum.LoopyMonteCarloSampling` method), 91  
`messageApproximationScheme()` (`pyAgrum.LoopyWeightedSampling` method), 97  
`messageApproximationScheme()` (`pyAgrum.MonteCarloSampling` method), 66  
`messageApproximationScheme()` (`pyAgrum.`

*grum.WeightedSampling method*), 72  
 min() (*pyAgrum.Potential method*), 152  
 minEpsilonRate() (*pyAgrum.BN Learner method*), 109  
 minEpsilonRate() (*pyAgrum.CN Loopy Propagation method*), 204  
 minEpsilonRate() (*pyAgrum.CN Monte Carlo Sampling method*), 202  
 minEpsilonRate() (*pyAgrum.Gibbs BN distance method*), 19  
 minEpsilonRate() (*pyAgrum.Gibbs Sampling method*), 60  
 minEpsilonRate() (*pyAgrum.Importance Sampling method*), 78  
 minEpsilonRate() (*pyAgrum.Loopy Belief Propagation method*), 54  
 minEpsilonRate() (*pyAgrum.Loopy Gibbs Sampling method*), 85  
 minEpsilonRate() (*pyAgrum.Loopy Importance Sampling method*), 103  
 minEpsilonRate() (*pyAgrum.Loopy Monte Carlo Sampling method*), 91  
 minEpsilonRate() (*pyAgrum.Loopy Weighted Sampling method*), 97  
 minEpsilonRate() (*pyAgrum.Monte Carlo Sampling method*), 66  
 minEpsilonRate() (*pyAgrum.Weighted Sampling method*), 72  
 minimalCondSet() (*pyAgrum.Bayes Net method*), 13  
 minimalCondSet() (*pyAgrum.Bayes Net Fragment method*), 28  
 minNonZero() (*pyAgrum.Potential method*), 153  
 minNonZeroParam() (*pyAgrum.Bayes Net method*), 13  
 minNonZeroParam() (*pyAgrum.Bayes Net Fragment method*), 28  
 minParam() (*pyAgrum.Bayes Net method*), 13  
 minParam() (*pyAgrum.Bayes Net Fragment method*), 28  
 minVal() (*pyAgrum.Range Variable method*), 140  
 MixedGraph (class in *pyAgrum*), 126  
 mixedGraph() (*pyAgrum.Essential Graph method*), 22  
 mixedOrientedPath() (*pyAgrum.Mixed Graph method*), 129  
 mixedUnorientedPath() (*pyAgrum.Mixed Graph method*), 129  
 MonteCarloSampling (class in *pyAgrum*), 62  
 moralGraph() (*pyAgrum.Bayes Net method*), 13  
 moralGraph() (*pyAgrum.Bayes Net Fragment method*), 28

moralGraph() (*pyAgrum.DAG method*), 119  
 moralGraph() (*pyAgrum.Influence Diagram method*), 211

## N

name() (*pyAgrum.Discrete Variable method*), 132  
 name() (*pyAgrum.Discretized Variable method*), 137  
 name() (*pyAgrum.Labelized Variable method*), 135  
 name() (*pyAgrum.Range Variable method*), 140  
 nameFromId() (*pyAgrum.BN Learner method*), 109  
 names() (*pyAgrum.Bayes Net method*), 14  
 names() (*pyAgrum.Bayes Net Fragment method*), 28  
 names() (*pyAgrum.BN Learner method*), 110  
 names() (*pyAgrum.causal.Causal Model method*), 180  
 names() (*pyAgrum.Influence Diagram method*), 211  
 nbCols() (*pyAgrum.BN Learner method*), 110  
 nbrDim() (*pyAgrum.Instantiation method*), 146  
 nbrDim() (*pyAgrum.Potential method*), 153  
 nbrDrawnVar() (*pyAgrum.Gibbs BN distance method*), 19  
 nbrDrawnVar() (*pyAgrum.Gibbs Sampling method*), 60  
 nbrDrawnVar() (*pyAgrum.Loopy Gibbs Sampling method*), 85  
 nbrEvidence() (*pyAgrum.Gibbs Sampling method*), 60  
 nbrEvidence() (*pyAgrum.Importance Sampling method*), 78  
 nbrEvidence() (*pyAgrum.Lazy Propagation method*), 35  
 nbrEvidence() (*pyAgrum.Loopy Belief Propagation method*), 54  
 nbrEvidence() (*pyAgrum.Loopy Gibbs Sampling method*), 85  
 nbrEvidence() (*pyAgrum.Loopy Importance Sampling method*), 103  
 nbrEvidence() (*pyAgrum.Loopy Monte Carlo Sampling method*), 91  
 nbrEvidence() (*pyAgrum.Loopy Weighted Sampling method*), 97  
 nbrEvidence() (*pyAgrum.Monte Carlo Sampling method*), 66  
 nbrEvidence() (*pyAgrum.Shafer Shenoy Inference method*), 42  
 nbrEvidence() (*pyAgrum.Variable Elimination method*), 48  
 nbrEvidence() (*pyAgrum.Weighted Sampling method*), 72  
 nbrHardEvidence() (*pyAgrum.Gibbs Sampling method*), 60  
 nbrHardEvidence() (*pyAgrum.Importance Sampling method*), 79  
 nbrHardEvidence() (*pyAgrum.Lazy Propagation method*), 35



`nbrHardEvidence()` (`pyAgrum.LoopBeliefPropagation` method), 54  
`nbrHardEvidence()` (`pyAgrum.LoopGibbsSampling` method), 85  
`nbrHardEvidence()` (`pyAgrum.LoopImportanceSampling` method), 103  
`nbrHardEvidence()` (`pyAgrum.LoopMonteCarloSampling` method), 91  
`nbrHardEvidence()` (`pyAgrum.LoopWeightedSampling` method), 97  
`nbrHardEvidence()` (`pyAgrum.MonteCarloSampling` method), 66  
`nbrHardEvidence()` (`pyAgrum.ShaferShenoyInference` method), 42  
`nbrHardEvidence()` (`pyAgrum.VariableElimination` method), 48  
`nbrHardEvidence()` (`pyAgrum.WeightedSampling` method), 72  
`nbrIterations()` (`pyAgrum.BN Learner` method), 110  
`nbrIterations()` (`pyAgrum.CN LoopyPropagation` method), 204  
`nbrIterations()` (`pyAgrum.CN MonteCarloSampling` method), 202  
`nbrIterations()` (`pyAgrum.GibbsBN distance` method), 19  
`nbrIterations()` (`pyAgrum.GibbsSampling` method), 60  
`nbrIterations()` (`pyAgrum.ImportanceSampling` method), 79  
`nbrIterations()` (`pyAgrum.LoopBeliefPropagation` method), 54  
`nbrIterations()` (`pyAgrum.LoopGibbsSampling` method), 85  
`nbrIterations()` (`pyAgrum.LoopImportanceSampling` method), 103  
`nbrIterations()` (`pyAgrum.LoopMonteCarloSampling` method), 91  
`nbrIterations()` (`pyAgrum.LoopWeightedSampling` method), 97  
`nbrIterations()` (`pyAgrum.MonteCarloSampling` method), 66  
`nbrIterations()` (`pyAgrum.WeightedSampling` method), 72  
`nbrJointTargets()` (`pyAgrum.LazyPropagation` method), 35  
`nbrJointTargets()` (`pyAgrum.ShaferShenoyInference` method), 42  
`nbrRows()` (`pyAgrum.BN Learner` method), 110  
`nbrSoftEvidence()` (`pyAgrum.GibbsSampling` method), 60  
`nbrSoftEvidence()` (`pyAgrum.ImportanceSampling` method), 79  
`nbrSoftEvidence()` (`pyAgrum.LazyPropagation` method), 35  
`nbrSoftEvidence()` (`pyAgrum.LoopBeliefPropagation` method), 54  
`nbrSoftEvidence()` (`pyAgrum.LoopGibbsSampling` method), 85  
`nbrSoftEvidence()` (`pyAgrum.LoopImportanceSampling` method), 104  
`nbrSoftEvidence()` (`pyAgrum.LoopMonteCarloSampling` method), 91  
`nbrSoftEvidence()` (`pyAgrum.LoopWeightedSampling` method), 97  
`nbrSoftEvidence()` (`pyAgrum.MonteCarloSampling` method), 66  
`nbrSoftEvidence()` (`pyAgrum.ShaferShenoyInference` method), 42  
`nbrSoftEvidence()` (`pyAgrum.VariableElimination` method), 48  
`nbrSoftEvidence()` (`pyAgrum.WeightedSampling` method), 73  
`nbrTargets()` (`pyAgrum.GibbsSampling` method), 60  
`nbrTargets()` (`pyAgrum.ImportanceSampling` method), 79  
`nbrTargets()` (`pyAgrum.LazyPropagation` method), 35  
`nbrTargets()` (`pyAgrum.LoopBeliefPropagation` method), 54  
`nbrTargets()` (`pyAgrum.LoopGibbsSampling` method), 85  
`nbrTargets()` (`pyAgrum.LoopImportanceSampling` method), 104  
`nbrTargets()` (`pyAgrum.LoopMonteCarloSampling` method), 91  
`nbrTargets()` (`pyAgrum.LoopWeightedSampling` method), 98  
`nbrTargets()` (`pyAgrum.MonteCarloSampling` method), 67  
`nbrTargets()` (`pyAgrum.ShaferShenoyInference` method), 42  
`nbrTargets()` (`pyAgrum.VariableElimination` method), 48  
`nbrTargets()` (`pyAgrum.WeightedSampling` method), 73

[neighbours\(\)](#) (*pyAgrum.CliqueGraph* method), 125  
[neighbours\(\)](#) (*pyAgrum.EssentialGraph* method), 22  
[neighbours\(\)](#) (*pyAgrum.MixedGraph* method), 129  
[neighbours\(\)](#) (*pyAgrum.UndiGraph* method), 121  
[newFactory\(\)](#) (*pyAgrum.Potential* method), 153  
[NoChild](#), 227  
[nodeId\(\)](#) (*pyAgrum.BayesNet* method), 14  
[nodeId\(\)](#) (*pyAgrum.BayesNetFragment* method), 28  
[nodeId\(\)](#) (*pyAgrum.InfluenceDiagram* method), 211  
[nodes\(\)](#) (*pyAgrum.BayesNet* method), 14  
[nodes\(\)](#) (*pyAgrum.BayesNetFragment* method), 28  
[nodes\(\)](#) (*pyAgrum.CliqueGraph* method), 125  
[nodes\(\)](#) (*pyAgrum.DAG* method), 119  
[nodes\(\)](#) (*pyAgrum.DiGraph* method), 117  
[nodes\(\)](#) (*pyAgrum.EssentialGraph* method), 23  
[nodes\(\)](#) (*pyAgrum.InfluenceDiagram* method), 212  
[nodes\(\)](#) (*pyAgrum.MarkovBlanket* method), 24  
[nodes\(\)](#) (*pyAgrum.MixedGraph* method), 129  
[nodes\(\)](#) (*pyAgrum.UndiGraph* method), 121  
[nodeType\(\)](#) (*pyAgrum.CredalNet* method), 199  
[noising\(\)](#) (*pyAgrum.Potential* method), 153  
[NoNeighbour](#), 227  
[NoParent](#), 227  
[normalize\(\)](#) (*pyAgrum.Potential* method), 153  
[normalizeAsCPT\(\)](#) (*pyAgrum.Potential* method), 153  
[NotFound](#), 228  
[NullElement](#), 228  
[numerical\(\)](#) (*pyAgrum.DiscreteVariable* method), 132  
[numerical\(\)](#) (*pyAgrum.DiscretizedVariable* method), 137  
[numerical\(\)](#) (*pyAgrum.LabelizedVariable* method), 135  
[numerical\(\)](#) (*pyAgrum.RangeVariable* method), 140

## O

[observationalBN\(\)](#) (*pyAgrum.causal.CausalModel* method), 180  
[op1](#) (*pyAgrum.causal.ASTBinaryOp* attribute), 183  
[op1](#) (*pyAgrum.causal.ASTdiv* attribute), 184  
[op1](#) (*pyAgrum.causal.ASTminus* attribute), 184  
[op1](#) (*pyAgrum.causal.ASTmult* attribute), 185  
[op1](#) (*pyAgrum.causal.ASTplus* attribute), 183  
[op2](#) (*pyAgrum.causal.ASTBinaryOp* attribute), 183  
[op2](#) (*pyAgrum.causal.ASTdiv* attribute), 184  
[op2](#) (*pyAgrum.causal.ASTminus* attribute), 184  
[op2](#) (*pyAgrum.causal.ASTmult* attribute), 185  
[op2](#) (*pyAgrum.causal.ASTplus* attribute), 183  
[OperationNotAllowed](#), 228  
[other\(\)](#) (*pyAgrum.Arc* method), 113  
[other\(\)](#) (*pyAgrum.Edge* method), 114  
[OutOfBounds](#), 229  
[OutOfLowerBound](#), 229

[OutOfUpperBound](#), 229

## P

[parents\(\)](#) (*pyAgrum.BayesNet* method), 14  
[parents\(\)](#) (*pyAgrum.BayesNetFragment* method), 28  
[parents\(\)](#) (*pyAgrum.causal.CausalModel* method), 180  
[parents\(\)](#) (*pyAgrum.DAG* method), 119  
[parents\(\)](#) (*pyAgrum.DiGraph* method), 117  
[parents\(\)](#) (*pyAgrum.EssentialGraph* method), 23  
[parents\(\)](#) (*pyAgrum.InfluenceDiagram* method), 212  
[parents\(\)](#) (*pyAgrum.MarkovBlanket* method), 24  
[parents\(\)](#) (*pyAgrum.MixedGraph* method), 129  
[partialUndiGraph\(\)](#) (*pyAgrum.CliqueGraph* method), 125  
[partialUndiGraph\(\)](#) (*pyAgrum.MixedGraph* method), 129  
[partialUndiGraph\(\)](#) (*pyAgrum.UndiGraph* method), 121  
[pdfize\(\)](#) (in module *pyAgrum.lib.bn2graph*), 170, 172  
[periodSize\(\)](#) (*pyAgrum.BN Learner* method), 110  
[periodSize\(\)](#) (*pyAgrum.CN LoopyPropagation* method), 204  
[periodSize\(\)](#) (*pyAgrum.CN Monte Carlo Sampling* method), 202  
[periodSize\(\)](#) (*pyAgrum.Gibbs BN distance* method), 19  
[periodSize\(\)](#) (*pyAgrum.Gibbs Sampling* method), 60  
[periodSize\(\)](#) (*pyAgrum.Importance Sampling* method), 79  
[periodSize\(\)](#) (*pyAgrum.Loopy Belief Propagation* method), 54  
[periodSize\(\)](#) (*pyAgrum.Loopy Gibbs Sampling* method), 85  
[periodSize\(\)](#) (*pyAgrum.Loopy Importance Sampling* method), 104  
[periodSize\(\)](#) (*pyAgrum.Loopy Monte Carlo Sampling* method), 91  
[periodSize\(\)](#) (*pyAgrum.Loopy Weighted Sampling* method), 98  
[periodSize\(\)](#) (*pyAgrum.Monte Carlo Sampling* method), 67  
[periodSize\(\)](#) (*pyAgrum.Weighted Sampling* method), 73  
[pngize\(\)](#) (in module *pyAgrum.lib.bn2graph*), 170, 172  
[populate\(\)](#) (*pyAgrum.Potential* method), 153  
[pos\(\)](#) (*pyAgrum.Instantiation* method), 146  
[pos\(\)](#) (*pyAgrum.Potential* method), 153  
[posLabel\(\)](#) (*pyAgrum.LabelizedVariable* method), 135

`posterior()` (*pyAgrum.GibbsSampling method*), 61  
`posterior()` (*pyAgrum.ImportanceSampling method*), 79  
`posterior()` (*pyAgrum.LazyPropagation method*), 36  
`posterior()` (*pyAgrum.LoopyBeliefPropagation method*), 54  
`posterior()` (*pyAgrum.LoopyGibbsSampling method*), 85  
`posterior()` (*pyAgrum.LoopyImportanceSampling method*), 104  
`posterior()` (*pyAgrum.LoopyMonteCarloSampling method*), 92  
`posterior()` (*pyAgrum.LoopyWeightedSampling method*), 98  
`posterior()` (*pyAgrum.MonteCarloSampling method*), 67  
`posterior()` (*pyAgrum.ShaferShenoyInference method*), 42  
`posterior()` (*pyAgrum.VariableElimination method*), 48  
`posterior()` (*pyAgrum.WeightedSampling method*), 73  
`Potential` (class in *pyAgrum*), 149  
`PRMexplorer` (class in *pyAgrum*), 189  
`proba2histo()` (in module *pyAgrum.lib.bn2graph*), 170, 171  
`product()` (*pyAgrum.Potential method*), 153  
`property()` (*pyAgrum.BayesNet method*), 14  
`property()` (*pyAgrum.BayesNetFragment method*), 28  
`property()` (*pyAgrum.InfluenceDiagram method*), 212  
`propertyWithDefault()` (*pyAgrum.BayesNet method*), 14  
`propertyWithDefault()` (*pyAgrum.BayesNetFragment method*), 29  
`propertyWithDefault()` (*pyAgrum.InfluenceDiagram method*), 212  
`putFirst()` (*pyAgrum.Potential method*), 153  
`pyAgrum.causal.notebook` (module), 186  
`pyAgrum.lib.bn2graph` (module), 169  
`pyAgrum.lib.notebook` (module), 157

## R

`random()` (*pyAgrum.Potential method*), 153  
`randomCPT()` (*pyAgrum.Potential method*), 153  
`randomDistribution()` (in module *pyAgrum*), 218  
`randomDistribution()` (*pyAgrum.Potential method*), 154  
`randomProba()` (in module *pyAgrum*), 218  
`RangeVariable` (class in *pyAgrum*), 138  
`recordWeight()` (*pyAgrum.BN Learner method*), 110  
`ReferenceError`, 230

`remainingBurnIn()` (*pyAgrum.GibbsBNdistance method*), 19  
`remove()` (*pyAgrum.Potential method*), 154  
`rend()` (*pyAgrum.Instantiation method*), 147  
`reorder()` (*pyAgrum.Instantiation method*), 147  
`reorganize()` (*pyAgrum.Potential method*), 154  
`reverseArc()` (*pyAgrum.BayesNet method*), 14  
`root` (*pyAgrum.causal.CausalFormula attribute*), 181

## S

`saveBIF()` (*pyAgrum.BayesNet method*), 14  
`saveBIFXML()` (*pyAgrum.BayesNet method*), 14  
`saveBIFXML()` (*pyAgrum.InfluenceDiagram method*), 212  
`saveBN()` (in module *pyAgrum*), 216  
`saveBNsMinMax()` (*pyAgrum.CredalNet method*), 199  
`saveDSL()` (*pyAgrum.BayesNet method*), 15  
`saveInference()` (*pyAgrum.CNLoopyPropagation method*), 204  
`saveNET()` (*pyAgrum.BayesNet method*), 15  
`saveO3PRM()` (*pyAgrum.BayesNet method*), 15  
`saveUAI()` (*pyAgrum.BayesNet method*), 15  
`scale()` (*pyAgrum.Potential method*), 154  
`second()` (*pyAgrum.Arc method*), 114  
`second()` (*pyAgrum.Edge method*), 114  
`separator()` (*pyAgrum.CliqueGraph method*), 125  
`set()` (*pyAgrum.Potential method*), 154  
`setAntiTopologicalVarOrder()` (*pyAgrum.BNDatabaseGenerator method*), 16  
`setAprioriWeight()` (*pyAgrum.BN Learner method*), 110  
`setBurnIn()` (*pyAgrum.GibbsBNdistance method*), 19  
`setBurnIn()` (*pyAgrum.GibbsSampling method*), 61  
`setBurnIn()` (*pyAgrum.LoopyGibbsSampling method*), 85  
`setClique()` (*pyAgrum.CliqueGraph method*), 125  
`setCPT()` (*pyAgrum.CredalNet method*), 199  
`setCPTs()` (*pyAgrum.CredalNet method*), 200  
`setDatabaseWeight()` (*pyAgrum.BN Learner method*), 110  
`setDescription()` (*pyAgrum.DiscreteVariable method*), 132  
`setDescription()` (*pyAgrum.DiscretizedVariable method*), 137  
`setDescription()` (*pyAgrum.LabelizedVariable method*), 135  
`setDescription()` (*pyAgrum.RangeVariable method*), 140  
`setDrawnAtRandom()` (*pyAgrum.GibbsBNdistance method*), 20  
`setDrawnAtRandom()` (*pyAgrum.GibbsSampling method*), 61  
`setDrawnAtRandom()` (*pyAgrum.LoopyGibbsSampling method*), 86

`setEpsilon()` (*pyAgrum.BN Learner method*), 110  
`setEpsilon()` (*pyAgrum.CN Loopy Propagation method*), 204  
`setEpsilon()` (*pyAgrum.CN Monte Carlo Sampling method*), 202  
`setEpsilon()` (*pyAgrum.Gibbs BN distance method*), 20  
`setEpsilon()` (*pyAgrum.Gibbs Sampling method*), 61  
`setEpsilon()` (*pyAgrum.Importance Sampling method*), 79  
`setEpsilon()` (*pyAgrum.Loopy Belief Propagation method*), 54  
`setEpsilon()` (*pyAgrum.Loopy Gibbs Sampling method*), 86  
`setEpsilon()` (*pyAgrum.Loopy Importance Sampling method*), 104  
`setEpsilon()` (*pyAgrum.Loopy Monte Carlo Sampling method*), 92  
`setEpsilon()` (*pyAgrum.Loopy Weighted Sampling method*), 98  
`setEpsilon()` (*pyAgrum.Monte Carlo Sampling method*), 67  
`setEpsilon()` (*pyAgrum.Weighted Sampling method*), 73  
`setEvidence()` (*pyAgrum.Gibbs Sampling method*), 61  
`setEvidence()` (*pyAgrum.Importance Sampling method*), 79  
`setEvidence()` (*pyAgrum.Influence Diagram Inference method*), 214  
`setEvidence()` (*pyAgrum.Lazy Propagation method*), 36  
`setEvidence()` (*pyAgrum.Loopy Belief Propagation method*), 55  
`setEvidence()` (*pyAgrum.Loopy Gibbs Sampling method*), 86  
`setEvidence()` (*pyAgrum.Loopy Importance Sampling method*), 104  
`setEvidence()` (*pyAgrum.Loopy Monte Carlo Sampling method*), 92  
`setEvidence()` (*pyAgrum.Loopy Weighted Sampling method*), 98  
`setEvidence()` (*pyAgrum.Monte Carlo Sampling method*), 67  
`setEvidence()` (*pyAgrum.Shafer Shenoy Inference method*), 43  
`setEvidence()` (*pyAgrum.Variable Elimination method*), 49  
`setEvidence()` (*pyAgrum.Weighted Sampling method*), 73  
`setFindBarrenNodesType()` (*pyAgrum.Lazy Propagation method*), 36  
`setFindBarrenNodesType()` (*pyAgrum.Shafer Shenoy Inference method*), 43  
`setFindBarrenNodesType()` (*pyAgrum.Variable Elimination method*), 49  
`setFirst()` (*pyAgrum.Instantiation method*), 147  
`setFirstIn()` (*pyAgrum.Instantiation method*), 147  
`setFirstNotVar()` (*pyAgrum.Instantiation method*), 147  
`setFirstOut()` (*pyAgrum.Instantiation method*), 147  
`setFirstVar()` (*pyAgrum.Instantiation method*), 147  
`setInitialDAG()` (*pyAgrum.BN Learner method*), 110  
`setLast()` (*pyAgrum.Instantiation method*), 147  
`setLastIn()` (*pyAgrum.Instantiation method*), 147  
`setLastNotVar()` (*pyAgrum.Instantiation method*), 147  
`setLastOut()` (*pyAgrum.Instantiation method*), 147  
`setLastVar()` (*pyAgrum.Instantiation method*), 148  
`setMaxIndegree()` (*pyAgrum.BN Learner method*), 110  
`setMaxIter()` (*pyAgrum.BN Learner method*), 110  
`setMaxIter()` (*pyAgrum.CN Loopy Propagation method*), 204  
`setMaxIter()` (*pyAgrum.CN Monte Carlo Sampling method*), 202  
`setMaxIter()` (*pyAgrum.Gibbs BN distance method*), 20  
`setMaxIter()` (*pyAgrum.Gibbs Sampling method*), 61  
`setMaxIter()` (*pyAgrum.Importance Sampling method*), 79  
`setMaxIter()` (*pyAgrum.Loopy Belief Propagation method*), 55  
`setMaxIter()` (*pyAgrum.Loopy Gibbs Sampling method*), 86  
`setMaxIter()` (*pyAgrum.Loopy Importance Sampling method*), 104  
`setMaxIter()` (*pyAgrum.Loopy Monte Carlo Sampling method*), 92  
`setMaxIter()` (*pyAgrum.Loopy Weighted Sampling method*), 98  
`setMaxIter()` (*pyAgrum.Monte Carlo Sampling method*), 67  
`setMaxIter()` (*pyAgrum.Weighted Sampling method*), 73  
`setMaxTime()` (*pyAgrum.BN Learner method*), 110  
`setMaxTime()` (*pyAgrum.CN Loopy Propagation method*), 205  
`setMaxTime()` (*pyAgrum.CN Monte Carlo Sampling*



method), 202

setMaxTime() (pyAgrum.GibbsBNdistance method), 20

setMaxTime() (pyAgrum.GibbsSampling method), 61

setMaxTime() (pyAgrum.ImportanceSampling method), 80

setMaxTime() (pyAgrum.LoopyBeliefPropagation method), 55

setMaxTime() (pyAgrum.LoopyGibbsSampling method), 86

setMaxTime() (pyAgrum.LoopyImportanceSampling method), 104

setMaxTime() (pyAgrum.LoopyMonteCarloSampling method), 92

setMaxTime() (pyAgrum.LoopyWeightedSampling method), 98

setMaxTime() (pyAgrum.MonteCarloSampling method), 67

setMaxTime() (pyAgrum.WeightedSampling method), 73

setMaxVal() (pyAgrum.RangeVariable method), 140

setMinEpsilonRate() (pyAgrum.BN Learner method), 110

setMinEpsilonRate() (pyAgrum.CN LoopyPropagation method), 205

setMinEpsilonRate() (pyAgrum.CN MonteCarloSampling method), 202

setMinEpsilonRate() (pyAgrum.GibbsBNdistance method), 20

setMinEpsilonRate() (pyAgrum.GibbsSampling method), 61

setMinEpsilonRate() (pyAgrum.ImportanceSampling method), 80

setMinEpsilonRate() (pyAgrum.LoopyBeliefPropagation method), 55

setMinEpsilonRate() (pyAgrum.LoopyGibbsSampling method), 86

setMinEpsilonRate() (pyAgrum.LoopyImportanceSampling method), 104

setMinEpsilonRate() (pyAgrum.LoopyMonteCarloSampling method), 92

setMinEpsilonRate() (pyAgrum.LoopyWeightedSampling method), 98

setMinEpsilonRate() (pyAgrum.MonteCarloSampling method), 67

setMinEpsilonRate() (pyAgrum.WeightedSampling method), 73

setMinVal() (pyAgrum.RangeVariable method), 140

setName() (pyAgrum.DiscreteVariable method), 132

setName() (pyAgrum.DiscretizedVariable method), 137

setName() (pyAgrum.LabelizedVariable method), 135

setName() (pyAgrum.RangeVariable method), 140

setNbrDrawnVar() (pyAgrum.GibbsBNdistance method), 20

setNbrDrawnVar() (pyAgrum.GibbsSampling method), 61

setNbrDrawnVar() (pyAgrum.LoopyGibbsSampling method), 86

setNumberOfThreads() (in module pyAgrum), 219

setPeriodSize() (pyAgrum.BN Learner method), 110

setPeriodSize() (pyAgrum.CN LoopyPropagation method), 205

setPeriodSize() (pyAgrum.CN MonteCarloSampling method), 202

setPeriodSize() (pyAgrum.GibbsBNdistance method), 20

setPeriodSize() (pyAgrum.GibbsSampling method), 61

setPeriodSize() (pyAgrum.ImportanceSampling method), 80

setPeriodSize() (pyAgrum.LoopyBeliefPropagation method), 55

setPeriodSize() (pyAgrum.LoopyGibbsSampling method), 86

setPeriodSize() (pyAgrum.LoopyImportanceSampling method), 105

setPeriodSize() (pyAgrum.LoopyMonteCarloSampling method), 92

setPeriodSize() (pyAgrum.LoopyWeightedSampling method), 98

setPeriodSize() (pyAgrum.MonteCarloSampling method), 67

setPeriodSize() (pyAgrum.WeightedSampling method), 74

setPossibleSkeleton() (pyAgrum.BN Learner method), 111

setProperty() (pyAgrum.BayesNet method), 15

setProperty() (pyAgrum.BayesNetFragment method), 29

setProperty() (pyAgrum.InfluenceDiagram method), 212

setRandomVarOrder() (pyAgrum.BN DatabaseGenerator method), 16

setRecordWeight() (pyAgrum.BN Learner

method), 111

setRelevantPotentialsFinderType() (pyAgrum.LazyPropagation method), 36

setRelevantPotentialsFinderType() (pyAgrum.VariableElimination method), 49

setRepetitiveInd() (pyAgrum.CNLoopyPropagation method), 205

setRepetitiveInd() (pyAgrum.CNMonteCarloSampling method), 202

setSliceOrder() (pyAgrum.BN Learner method), 111

setTargets() (pyAgrum.GibbsSampling method), 61

setTargets() (pyAgrum.ImportanceSampling method), 80

setTargets() (pyAgrum.LazyPropagation method), 36

setTargets() (pyAgrum.LoopyBeliefPropagation method), 55

setTargets() (pyAgrum.LoopyGibbsSampling method), 86

setTargets() (pyAgrum.LoopyImportanceSampling method), 105

setTargets() (pyAgrum.LoopyMonteCarloSampling method), 92

setTargets() (pyAgrum.LoopyWeightedSampling method), 98

setTargets() (pyAgrum.MonteCarloSampling method), 68

setTargets() (pyAgrum.ShaferShenoyInference method), 43

setTargets() (pyAgrum.VariableElimination method), 49

setTargets() (pyAgrum.WeightedSampling method), 74

setTopologicalVarOrder() (pyAgrum.BNDatabaseGenerator method), 16

setTriangulation() (pyAgrum.LazyPropagation method), 37

setTriangulation() (pyAgrum.ShaferShenoyInference method), 43

setTriangulation() (pyAgrum.VariableElimination method), 49

setVals() (pyAgrum.Instantiation method), 148

setVarOrder() (pyAgrum.BNDatabaseGenerator method), 16

setVarOrderFromCSV() (pyAgrum.BNDatabaseGenerator method), 16

setVerbosity() (pyAgrum.BN Learner method), 111

setVerbosity() (pyAgrum.CNLoopyPropagation method), 205

setVerbosity() (pyAgrum.CNMonteCarloSampling method), 202

setVerbosity() (pyAgrum.GibbsBNdistance method), 20

setVerbosity() (pyAgrum.GibbsSampling method), 62

setVerbosity() (pyAgrum.ImportanceSampling method), 80

setVerbosity() (pyAgrum.LoopyBeliefPropagation method), 55

setVerbosity() (pyAgrum.LoopyGibbsSampling method), 86

setVerbosity() (pyAgrum.LoopyImportanceSampling method), 105

setVerbosity() (pyAgrum.LoopyMonteCarloSampling method), 92

setVerbosity() (pyAgrum.LoopyWeightedSampling method), 99

setVerbosity() (pyAgrum.MonteCarloSampling method), 68

setVerbosity() (pyAgrum.WeightedSampling method), 74

setVirtualLBPSize() (pyAgrum.LoopyGibbsSampling method), 86

setVirtualLBPSize() (pyAgrum.LoopyImportanceSampling method), 105

setVirtualLBPSize() (pyAgrum.LoopyMonteCarloSampling method), 92

setVirtualLBPSize() (pyAgrum.LoopyWeightedSampling method), 99

ShaferShenoyInference (class in pyAgrum), 37

showBN() (in module pyAgrum.lib.notebook), 160, 165

showBNDiff() (in module pyAgrum.lib.notebook), 160

showCausalImpact() (in module pyAgrum.causal.notebook), 187

showCausalModel() (in module pyAgrum.causal.notebook), 187

showDot() (in module pyAgrum.lib.notebook), 160, 164

showGraph() (in module pyAgrum.lib.notebook), 160, 164

showInference() (in module pyAgrum.lib.notebook), 160, 165

showInfluenceDiagram() (in module pyAgrum.lib.notebook), 161, 167

showInformation() (in module pyAgrum.lib.notebook), 161, 166

`showJunctionTree()` (in module `pyAgrum.lib.notebook`), 161, 166  
`showPosterior()` (in module `pyAgrum.lib.notebook`), 161, 163  
`showPotential()` (in module `pyAgrum.lib.notebook`), 162, 163  
`showProba()` (in module `pyAgrum.lib.notebook`), 162, 163  
`sideBySide()` (in module `pyAgrum.lib.notebook`), 162, 163  
`size()` (`pyAgrum.BayesNet` method), 15  
`size()` (`pyAgrum.BayesNetFragment` method), 29  
`size()` (`pyAgrum.CliqueGraph` method), 125  
`size()` (`pyAgrum.DAG` method), 119  
`size()` (`pyAgrum.DiGraph` method), 117  
`size()` (`pyAgrum.EssentialGraph` method), 23  
`size()` (`pyAgrum.InfluenceDiagram` method), 212  
`size()` (`pyAgrum.MarkovBlanket` method), 24  
`size()` (`pyAgrum.MixedGraph` method), 129  
`size()` (`pyAgrum.UndiGraph` method), 122  
`sizeArcs()` (`pyAgrum.BayesNet` method), 15  
`sizeArcs()` (`pyAgrum.BayesNetFragment` method), 29  
`sizeArcs()` (`pyAgrum.DAG` method), 119  
`sizeArcs()` (`pyAgrum.DiGraph` method), 117  
`sizeArcs()` (`pyAgrum.EssentialGraph` method), 23  
`sizeArcs()` (`pyAgrum.InfluenceDiagram` method), 212  
`sizeArcs()` (`pyAgrum.MarkovBlanket` method), 24  
`sizeArcs()` (`pyAgrum.MixedGraph` method), 129  
`sizeEdges()` (`pyAgrum.CliqueGraph` method), 125  
`sizeEdges()` (`pyAgrum.EssentialGraph` method), 23  
`sizeEdges()` (`pyAgrum.MixedGraph` method), 130  
`sizeEdges()` (`pyAgrum.UndiGraph` method), 122  
`SizeError`, 230  
`sizeNodes()` (`pyAgrum.EssentialGraph` method), 23  
`sizeNodes()` (`pyAgrum.MarkovBlanket` method), 24  
`skeleton()` (`pyAgrum.EssentialGraph` method), 23  
`softEvidenceNodes()` (`pyAgrum.GibbsSampling` method), 62  
`softEvidenceNodes()` (`pyAgrum.ImportanceSampling` method), 80  
`softEvidenceNodes()` (`pyAgrum.LazyPropagation` method), 37  
`softEvidenceNodes()` (`pyAgrum.LoopyBeliefPropagation` method), 55  
`softEvidenceNodes()` (`pyAgrum.LoopyGibbsSampling` method), 86  
`softEvidenceNodes()` (`pyAgrum.LoopyImportanceSampling` method), 105  
`softEvidenceNodes()` (`pyAgrum.LoopyMonteCarloSampling` method), 93  
`softEvidenceNodes()` (`pyAgrum.LoopyWeightedSampling` method), 99  
`softEvidenceNodes()` (`pyAgrum.MonteCarloSampling` method), 68  
`softEvidenceNodes()` (`pyAgrum.ShaferShenoyInference` method), 43  
`softEvidenceNodes()` (`pyAgrum.VariableElimination` method), 49  
`softEvidenceNodes()` (`pyAgrum.WeightedSampling` method), 74  
`sq()` (`pyAgrum.Potential` method), 154  
`src_bn()` (`pyAgrum.CredalNet` method), 200  
`startOfPeriod()` (`pyAgrum.GibbsBNdistance` method), 20  
`stateApproximationScheme()` (`pyAgrum.GibbsBNdistance` method), 20  
`stopApproximationScheme()` (`pyAgrum.GibbsBNdistance` method), 20  
`sum()` (`pyAgrum.Potential` method), 154  
`SyntaxError`, 230

## T

`tail()` (`pyAgrum.Arc` method), 114  
`targets()` (`pyAgrum.GibbsSampling` method), 62  
`targets()` (`pyAgrum.ImportanceSampling` method), 80  
`targets()` (`pyAgrum.LazyPropagation` method), 37  
`targets()` (`pyAgrum.LoopyBeliefPropagation` method), 55  
`targets()` (`pyAgrum.LoopyGibbsSampling` method), 86  
`targets()` (`pyAgrum.LoopyImportanceSampling` method), 105  
`targets()` (`pyAgrum.LoopyMonteCarloSampling` method), 93  
`targets()` (`pyAgrum.LoopyWeightedSampling` method), 99  
`targets()` (`pyAgrum.MonteCarloSampling` method), 68  
`targets()` (`pyAgrum.ShaferShenoyInference` method), 43  
`targets()` (`pyAgrum.VariableElimination` method), 49  
`targets()` (`pyAgrum.WeightedSampling` method), 74  
`tick()` (`pyAgrum.DiscretizedVariable` method), 137  
`ticks()` (`pyAgrum.DiscretizedVariable` method), 138  
`toarray()` (`pyAgrum.Potential` method), 154  
`toBN()` (`pyAgrum.BayesNetFragment` method), 29  
`toCSV()` (`pyAgrum.BNDatabaseGenerator` method), 16  
`toDatabaseTable()` (`pyAgrum.BNDatabaseGenerator` method), 17  
`todict()` (`pyAgrum.Instantiation` method), 148  
`toDiscretizedVar()` (`pyAgrum.DiscreteVariable` method), 132

`toDiscretizedVar()` (`pyAgrum.DiscretizedVariable` method), 138  
`toDiscretizedVar()` (`pyAgrum.LabelizedVariable` method), 135  
`toDiscretizedVar()` (`pyAgrum.RangeVariable` method), 140  
`toDot()` (`pyAgrum.BayesNet` method), 15  
`toDot()` (`pyAgrum.BayesNetFragment` method), 29  
`toDot()` (`pyAgrum.CliqueGraph` method), 125  
`toDot()` (`pyAgrum.DAG` method), 119  
`toDot()` (`pyAgrum.DiGraph` method), 117  
`toDot()` (`pyAgrum.EssentialGraph` method), 23  
`toDot()` (`pyAgrum.InfluenceDiagram` method), 212  
`toDot()` (`pyAgrum.MarkovBlanket` method), 24  
`toDot()` (`pyAgrum.MixedGraph` method), 130  
`toDot()` (`pyAgrum.UndiGraph` method), 122  
`toDotWithNames()` (`pyAgrum.CliqueGraph` method), 126  
`toLabelizedVar()` (`pyAgrum.DiscreteVariable` method), 132  
`toLabelizedVar()` (`pyAgrum.DiscretizedVariable` method), 138  
`toLabelizedVar()` (`pyAgrum.LabelizedVariable` method), 135  
`toLabelizedVar()` (`pyAgrum.RangeVariable` method), 141  
`toLatex()` (`pyAgrum.causal.ASTBinaryOp` method), 183  
`toLatex()` (`pyAgrum.causal.ASTdiv` method), 184  
`toLatex()` (`pyAgrum.causal.ASTjointProba` method), 185  
`toLatex()` (`pyAgrum.causal.ASTminus` method), 184  
`toLatex()` (`pyAgrum.causal.ASTmult` method), 185  
`toLatex()` (`pyAgrum.causal.ASTplus` method), 183  
`toLatex()` (`pyAgrum.causal.ASTposteriorProba` method), 186  
`toLatex()` (`pyAgrum.causal.ASTsum` method), 185  
`toLatex()` (`pyAgrum.causal.ASTtree` method), 183  
`toLatex()` (`pyAgrum.causal.CausalFormula` method), 181  
`tolist()` (`pyAgrum.Potential` method), 154  
`topologicalOrder()` (`pyAgrum.BayesNet` method), 15  
`topologicalOrder()` (`pyAgrum.BayesNetFragment` method), 29  
`topologicalOrder()` (`pyAgrum.DAG` method), 119  
`topologicalOrder()` (`pyAgrum.DiGraph` method), 117  
`topologicalOrder()` (`pyAgrum.InfluenceDiagram` method), 212  
`topologicalOrder()` (`pyAgrum.MixedGraph` method), 130  
`toRangeVar()` (`pyAgrum.DiscreteVariable` method), 132  
`toRangeVar()` (`pyAgrum.DiscretizedVariable` method), 138  
`toRangeVar()` (`pyAgrum.LabelizedVariable` method), 135  
`toRangeVar()` (`pyAgrum.RangeVariable` method), 141  
`toStringWithDescription()` (`pyAgrum.DiscreteVariable` method), 132  
`toStringWithDescription()` (`pyAgrum.DiscretizedVariable` method), 138  
`toStringWithDescription()` (`pyAgrum.LabelizedVariable` method), 135  
`toStringWithDescription()` (`pyAgrum.RangeVariable` method), 141  
`translate()` (`pyAgrum.Potential` method), 154  
`type` (`pyAgrum.causal.ASTBinaryOp` attribute), 183  
`type` (`pyAgrum.causal.ASTdiv` attribute), 184  
`type` (`pyAgrum.causal.ASTjointProba` attribute), 185  
`type` (`pyAgrum.causal.ASTminus` attribute), 184  
`type` (`pyAgrum.causal.ASTmult` attribute), 185  
`type` (`pyAgrum.causal.ASTplus` attribute), 184  
`type` (`pyAgrum.causal.ASTposteriorProba` attribute), 186  
`type` (`pyAgrum.causal.ASTsum` attribute), 185  
`type` (`pyAgrum.causal.ASTtree` attribute), 183  
`types()` (`pyAgrum.PRMexplorer` method), 193

## U

`UndefinedElement`, 231  
`UndefinedIteratorKey`, 231  
`UndefinedIteratorValue`, 232  
`UndiGraph` (class in `pyAgrum`), 119  
`UnidentifiableException` (class in `pyAgrum.causal`), 186  
`uninstallCPT()` (`pyAgrum.BayesNetFragment` method), 29  
`uninstallNode()` (`pyAgrum.BayesNetFragment` method), 29  
`UnknownLabelInDatabase`, 232  
`unsetEnd()` (`pyAgrum.Instantiation` method), 148  
`unsetOverflow()` (`pyAgrum.Instantiation` method), 148  
`updateApproximationScheme()` (`pyAgrum.GibbsBNDistance` method), 20  
`updateEvidence()` (`pyAgrum.GibbsSampling` method), 62  
`updateEvidence()` (`pyAgrum.ImportanceSampling` method), 80  
`updateEvidence()` (`pyAgrum.LazyPropagation` method), 37  
`updateEvidence()` (`pyAgrum.LoopyBeliefPropagation` method), 55  
`updateEvidence()` (`pyAgrum.LoopyGibbsSampling` method), 87  
`updateEvidence()` (`pyAgrum.LoopyImportanceSampling` method), 105  
`updateEvidence()` (`pyAgrum.LoopyMonteCarloSampling` method),



- 93
- `updateEvidence()` (*pyAgrum.LoopyWeightedSampling method*), 99
- `updateEvidence()` (*pyAgrum.MonteCarloSampling method*), 68
- `updateEvidence()` (*pyAgrum.ShaferShenoyInference method*), 43
- `updateEvidence()` (*pyAgrum.VariableElimination method*), 50
- `updateEvidence()` (*pyAgrum.WeightedSampling method*), 74
- `use3off2()` (*pyAgrum.BN Learner method*), 111
- `useAprioriBDeu()` (*pyAgrum.BN Learner method*), 111
- `useAprioriDirichlet()` (*pyAgrum.BN Learner method*), 111
- `useAprioriSmoothing()` (*pyAgrum.BN Learner method*), 111
- `useEM()` (*pyAgrum.BN Learner method*), 111
- `useGreedyHillClimbing()` (*pyAgrum.BN Learner method*), 111
- `useK2()` (*pyAgrum.BN Learner method*), 111
- `useLocalSearchWithTabuList()` (*pyAgrum.BN Learner method*), 111
- `useMDL()` (*pyAgrum.BN Learner method*), 111
- `useMIIC()` (*pyAgrum.BN Learner method*), 111
- `useNML()` (*pyAgrum.BN Learner method*), 111
- `useNoApriori()` (*pyAgrum.BN Learner method*), 112
- `useNoCorr()` (*pyAgrum.BN Learner method*), 112
- `useScoreAIC()` (*pyAgrum.BN Learner method*), 112
- `useScoreBD()` (*pyAgrum.BN Learner method*), 112
- `useScoreBDeu()` (*pyAgrum.BN Learner method*), 112
- `useScoreBIC()` (*pyAgrum.BN Learner method*), 112
- `useScoreK2()` (*pyAgrum.BN Learner method*), 112
- `useScoreLog2Likelihood()` (*pyAgrum.BN Learner method*), 112
- `utility()` (*pyAgrum.InfluenceDiagram method*), 212
- `utilityNodeSize()` (*pyAgrum.InfluenceDiagram method*), 213
- V**
- `val()` (*pyAgrum.Instantiation method*), 148
- `var_dims` (*pyAgrum.Potential attribute*), 155
- `var_names` (*pyAgrum.Potential attribute*), 155
- `variable()` (*pyAgrum.BayesNet method*), 15
- `variable()` (*pyAgrum.BayesNetFragment method*), 29
- `variable()` (*pyAgrum.InfluenceDiagram method*), 213
- `variable()` (*pyAgrum.Instantiation method*), 148
- `variable()` (*pyAgrum.Potential method*), 155
- `VariableElimination` (*class in pyAgrum*), 44
- `variableFromName()` (*pyAgrum.BayesNet method*), 16
- `variableFromName()` (*pyAgrum.BayesNetFragment method*), 30
- `variableFromName()` (*pyAgrum.InfluenceDiagram method*), 213
- `variableNodeMap()` (*pyAgrum.BayesNet method*), 16
- `variableNodeMap()` (*pyAgrum.BayesNetFragment method*), 30
- `variableNodeMap()` (*pyAgrum.InfluenceDiagram method*), 213
- `variablesSequence()` (*pyAgrum.Instantiation method*), 148
- `variablesSequence()` (*pyAgrum.Potential method*), 155
- `varNames` (*pyAgrum.causal.ASTjointProba attribute*), 185
- `varOrder()` (*pyAgrum.BN DatabaseGenerator method*), 17
- `varOrderNames()` (*pyAgrum.BN DatabaseGenerator method*), 17
- `vars` (*pyAgrum.causal.ASTposteriorProba attribute*), 186
- `varType()` (*pyAgrum.DiscreteVariable method*), 133
- `varType()` (*pyAgrum.DiscretizedVariable method*), 138
- `varType()` (*pyAgrum.LabelizedVariable method*), 135
- `varType()` (*pyAgrum.RangeVariable method*), 141
- `verbosity()` (*pyAgrum.BN Learner method*), 112
- `verbosity()` (*pyAgrum.CN LoopyPropagation method*), 205
- `verbosity()` (*pyAgrum.CN MonteCarloSampling method*), 202
- `verbosity()` (*pyAgrum.GibbsBNdistance method*), 20
- `verbosity()` (*pyAgrum.GibbsSampling method*), 62
- `verbosity()` (*pyAgrum.ImportanceSampling method*), 80
- `verbosity()` (*pyAgrum.LoopyBeliefPropagation method*), 56
- `verbosity()` (*pyAgrum.LoopyGibbsSampling method*), 87
- `verbosity()` (*pyAgrum.LoopyImportanceSampling method*), 105
- `verbosity()` (*pyAgrum.LoopyMonteCarloSampling method*), 93
- `verbosity()` (*pyAgrum.LoopyWeightedSampling method*), 99
- `verbosity()` (*pyAgrum.MonteCarloSampling method*), 68
- `verbosity()` (*pyAgrum.WeightedSampling method*), 74

VI () (*pyAgrum.LazyPropagation method*), 31  
 VI () (*pyAgrum.ShaferShenoyInference method*), 38

## W

WeightedSampling (*class in pyAgrum*), 68  
 what () (*pyAgrum.DefaultInLabel method*), 221  
 what () (*pyAgrum.DuplicateElement method*), 222  
 what () (*pyAgrum.DuplicateLabel method*), 222  
 what () (*pyAgrum.EmptyBSTree method*), 222  
 what () (*pyAgrum.EmptySet method*), 223  
 what () (*pyAgrum.FatalError method*), 223  
 what () (*pyAgrum.FormatNotFound method*), 224  
 what () (*pyAgrum.GraphError method*), 224  
 what () (*pyAgrum.GumException method*), 223  
 what () (*pyAgrum.IdError method*), 225  
 what () (*pyAgrum.InvalidArc method*), 225  
 what () (*pyAgrum.InvalidArgument method*), 225  
 what () (*pyAgrum.InvalidArgumentsNumber method*), 226  
 what () (*pyAgrum.InvalidDirectedCycle method*), 226  
 what () (*pyAgrum.InvalidEdge method*), 226  
 what () (*pyAgrum.InvalidNode method*), 227  
 what () (*pyAgrum.IOException method*), 224  
 what () (*pyAgrum.NoChild method*), 227  
 what () (*pyAgrum.NoNeighbour method*), 227  
 what () (*pyAgrum.NoParent method*), 228  
 what () (*pyAgrum.NotFound method*), 228  
 what () (*pyAgrum.NullElement method*), 228  
 what () (*pyAgrum.OperationNotAllowed method*), 229  
 what () (*pyAgrum.OutOfBounds method*), 229  
 what () (*pyAgrum.OutOfLowerBound method*), 229  
 what () (*pyAgrum.OutOfUpperBound method*), 230  
 what () (*pyAgrum.ReferenceError method*), 230  
 what () (*pyAgrum.SizeError method*), 230  
 what () (*pyAgrum.SyntaxError method*), 231  
 what () (*pyAgrum.UndefinedElement method*), 231  
 what () (*pyAgrum.UndefinedIteratorKey method*), 232  
 what () (*pyAgrum.UndefinedIteratorValue method*), 232  
 what () (*pyAgrum.UnknownLabelInDatabase method*), 232  
 whenArcAdded () (*pyAgrum.BayesNetFragment method*), 30  
 whenArcDeleted () (*pyAgrum.BayesNetFragment method*), 30  
 whenNodeAdded () (*pyAgrum.BayesNetFragment method*), 30  
 whenNodeDeleted () (*pyAgrum.BayesNetFragment method*), 30  
 with\_traceback () (*pyAgrum.causal.HedgeException method*), 186  
 with\_traceback () (*pyAgrum.causal.UnidentifiableException method*), 186

with\_traceback () (*pyAgrum.DefaultInLabel method*), 221  
 with\_traceback () (*pyAgrum.DuplicateElement method*), 222  
 with\_traceback () (*pyAgrum.DuplicateLabel method*), 222  
 with\_traceback () (*pyAgrum.EmptyBSTree method*), 222  
 with\_traceback () (*pyAgrum.EmptySet method*), 223  
 with\_traceback () (*pyAgrum.FatalError method*), 223  
 with\_traceback () (*pyAgrum.FormatNotFound method*), 224  
 with\_traceback () (*pyAgrum.GraphError method*), 224  
 with\_traceback () (*pyAgrum.GumException method*), 223  
 with\_traceback () (*pyAgrum.IdError method*), 225  
 with\_traceback () (*pyAgrum.InvalidArc method*), 225  
 with\_traceback () (*pyAgrum.InvalidArgument method*), 225  
 with\_traceback () (*pyAgrum.InvalidArgumentsNumber method*), 226  
 with\_traceback () (*pyAgrum.InvalidDirectedCycle method*), 226  
 with\_traceback () (*pyAgrum.InvalidEdge method*), 226  
 with\_traceback () (*pyAgrum.InvalidNode method*), 227  
 with\_traceback () (*pyAgrum.IOException method*), 224  
 with\_traceback () (*pyAgrum.NoChild method*), 227  
 with\_traceback () (*pyAgrum.NoNeighbour method*), 227  
 with\_traceback () (*pyAgrum.NoParent method*), 228  
 with\_traceback () (*pyAgrum.NotFound method*), 228  
 with\_traceback () (*pyAgrum.NullElement method*), 228  
 with\_traceback () (*pyAgrum.OperationNotAllowed method*), 229  
 with\_traceback () (*pyAgrum.OutOfBounds method*), 229  
 with\_traceback () (*pyAgrum.OutOfLowerBound method*), 229  
 with\_traceback () (*pyAgrum.OutOfUpperBound method*), 230  
 with\_traceback () (*pyAgrum.ReferenceError method*), 230  
 with\_traceback () (*pyAgrum.SizeError method*), 230  
 with\_traceback () (*pyAgrum.SyntaxError*

*method*), [231](#)  
`with_traceback()` (*pyAgrum.UndefinedElement*  
*method*), [231](#)  
`with_traceback()` (*pyA-*  
*grum.UndefinedIteratorKey method*), [232](#)  
`with_traceback()` (*pyA-*  
*grum.UndefinedIteratorValue method*),  
[232](#)  
`with_traceback()` (*pyA-*  
*grum.UnknownLabelInDatabase method*),  
[232](#)