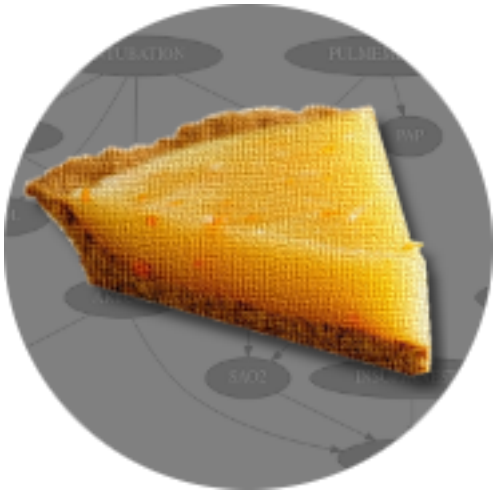


pyAgrum Documentation

Release 0.22.0

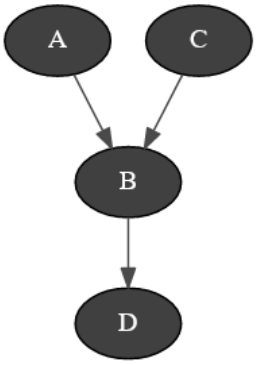
Pierre-Henri Wuillemin (Sphinx)

August 06, 2021

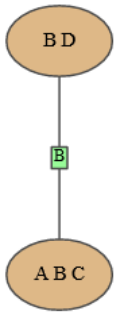
Fundamental components

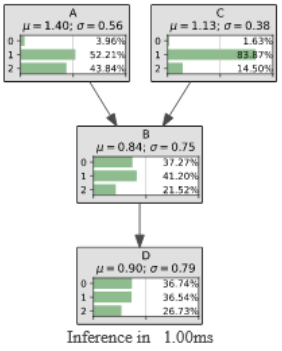
pyAgrum (http://agrum.org) is a scientific C++ and Python library dedicated to Bayesian networks (BN) and other Probabilistic Graphical Models. Based on the C++ aGrUM (https://agrum.lip6.fr) library, it provides a high-level interface to the C++ part of aGrUM allowing to create, manage and perform efficient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

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



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





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

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

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

pyAgrum contains

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

CHAPTER 1

Graphs manipulation

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

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

1.1 Edges and Arcs

1.1.1 Arc

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

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

Arc(tail, head) -> Arc

Parameters:

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

Arc(src) -> Arc

Parameters:

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

first(*self*)

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

Return type int

head(*self*)

Returns the id of the head node

Return type int

other(*self*, *id*)

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

Returns the nodeId of the other node

Return type int

second(*self*)

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

Return type int

tail(*self*)

Returns the id of the tail node

Return type int

1.1.2 Edge

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

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

Edge(aN1,aN2) -> Edge

Parameters:

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

Edge(src) -> Edge

Parameters:

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

first(*self*)

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

Return type int

other(*self*, *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(*self*)

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

Return type int

1.2 Directed Graphs

1.2.1 Digraph

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

DiGraph represents a Directed Graph.

DiGraph() -> DiGraph default constructor

DiGraph(src) -> DiGraph

Parameters:

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

addArc(*self*, *tail*, *head*)
addArc(self, n1, n2)

Add an arc from tail to head.

Parameters

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

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

addNode(*self*)

Returns the new NodeId

Return type int

addNodeWithId(*self*, *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(*self*, *n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

arcs(*self*)

Returns the list of the arcs

Return type List

children(*self*, *id*)

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

Returns the set of all the children

Return type Set

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

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

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

Returns True if the graph is empty

Return type bool

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

Returns True if the graph doesn't contains arcs

Return type bool

eraseArc(*self*, *n1*, *n2*)
Erase the arc between n1 and n2.

Parameters

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

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

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

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

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

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

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

existsArc(*self*, *n1*, *n2*)
Check if an arc exists bewteen n1 and n2.

Parameters

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

Returns True if the arc exists

Return type bool

existsNode(*self*, *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(*self*, *_from*, *to*)
Check if a directedpath exists bewteen from and to.

Parameters

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

Returns True if the directed path exists

Return type bool

nodes(*self*)

Returns the set of ids

Return type set

parents(*self*, *id*)

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

Returns the set of the parents ids.

Return type Set

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

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

Returns the list of the nodes Ids in a topological order

Return type List

Raises gum.InvalidDirectedCycle – If this graph contains cycles

1.2.2 Directed Acyclic Graph

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

DAG represents a Directed Acyclic Graph.

DAG() -> DAG default constructor

DAG(src) -> DAG

Parameters:

- **src** (*DAG*) – the DAG to copy

addArc(*self*, *tail*, *head*)

addArc(self, n1, n2)

Add an arc from tail to head.

Parameters

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

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(*self*)

Returns the new NodeId

Return type int

addNodeWithId(*self*, *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(*self*, *n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

arcs(*self*)

Returns the list of the arcs

Return type List

children(*self*, *id*)

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

Returns the set of all the children

Return type Set

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

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

dSeparation(*self*, *X*, *Y*, *Z*)
dSeparation(self, X, Y, Z) -> bool

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

Returns True if the graph is empty

Return type bool

emptyArcs(*self*)

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

eraseChildren(*self*, *n*)

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

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

eraseParents(*self*, *n*)

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

existsNode(*self*, *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 (*self*, *_from*, *to*)
Check if a directedpath exists bewteen from and to.

Parameters

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

Returns True if the directed path exists

Return type bool

moralGraph (*self*)

moralizedAncestralGraph (*self*, *nodes*)

nodes (*self*)

Returns the set of ids

Return type set

parents (*self*, *id*)

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

Returns the set of the parents ids.

Return type Set

size (*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs (*self*)

toDot (*self*)

Returns a friendly display of the graph in DOT format

Return type str

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

Returns the list of the nodes Ids in a topological order

Return type List

Raises gum.InvalidDirectedCycle – If this graph contains cycles

1.3 Undirected Graphs

1.3.1 UndiGraph

class pyAgrum.**UndiGraph** (**args*)
UndiGraph represents an Undirected Graph.

UndiGraph() -> UndiGraph default constructor

UndiGraph(src) -> UndiGraph

Parameters!

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

addEdge (*self*, *first*, *second*)
addEdge(self, n1, n2)

Insert a new edge into the graph.

Parameters

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

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

addNode(*self*)

Returns the new NodeId

Return type int

addNodeWithId(*self*, *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(*self*, *n*)
Add n nodes.

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

Returns the new ids

Return type Set of int

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

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

edges(*self*)

Returns the list of the edges

Return type List

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

Returns True if the graph is empty

Return type bool

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

Returns True if the graph doesn't contains edges

Return type bool

eraseEdge(*self*, *n1*, *n2*)
Erase the edge between n1 and n2.

Parameters

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

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

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

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

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

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

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

Parameters

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

Returns True if the arc exists

Return type bool

existsNode(*self*, *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(*self*)
Checks whether the graph contains cycles.

Returns True if the graph contains a cycle

Return type bool

neighbours(*self*, *id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*self*)

partialUndiGraph(*self*, *nodes*)

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

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeEdges(*self*)

Returns the number of edges in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

1.3.2 Clique Graph

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

CliqueGraph represents a Clique Graph.

CliqueGraph() -> CliqueGraph default constructor

CliqueGraph(src) -> CliqueGraph

Parameter

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

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

Insert a new edge into the graph.

Parameters

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

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

addNode(*self*, *clique*)

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

Returns the new NodeId

Return type int

addNodeWithId(*self*, *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(*self*, *n*)

Add n nodes.

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

Returns the new ids

Return type Set of int

addToClique(*self*, *clique_id*, *node_id*)

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

Parameters

- **clique_id** (*int*) – the id of the clique
- **node_id** (*int*) – the id of the node

Raises

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

clear(*self*)

Remove all the nodes and edges from the graph.

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

clique(*self*, *clique*)

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

Returns The set of nodes included in the clique

Return type Set

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

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

container(*self*, *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(*self*, *node1*, *node2*)

Parameters

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

Returns a path from a clique containing node1 to a clique containing node2

Return type List

Raises gum.NotFound – If such path cannot be found

edges(*self*)

Returns the list of the edges

Return type List

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

Returns True if the graph is empty

Return type bool

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

Returns True if the graph doesn't contains edges

Return type bool

eraseEdge(*self*, *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(*self*, *clique_id*, *node_id*)
Remove a node from a clique

Parameters

- **clique_id** (*int*) – the id of the clique
- **node_id** (*int*) – the id of the node

Raises gum.NotFound – If clique_id does not exist

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

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

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

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

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

Parameters

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

Returns True if the arc exists

Return type bool

existsNode(*self*, *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(*self*)

Returns True if the running intersection property holds

Return type bool

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

Returns True if the graph contains a cycle

Return type bool

isJoinTree(*self*)

Returns True if the graph is a join tree

Return type bool

neighbours(*self*, *id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*self*)

partialUndiGraph(*self*, *nodes*)

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

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

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

Parameters

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

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(*self*, *idClique*, *new_clique*)
changes the set of nodes included into a given clique

Parameters

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

Raises gum.NotFound – If idClique is not a clique of the graph

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeEdges(*self*)

Returns the number of edges in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

toDotWithNames(*bn*)

Parameters

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

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

Return type str

1.4 Mixed Graph

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

MixedGraph represents a graph with both arcs and edges.

MixedGraph() -> MixedGraph default constructor

MixedGraph(src) -> MixedGraph

Parameters:

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

addArc(*self*, *n1*, *n2*)

Add an arc from tail to head.

Parameters

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

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

addEdge(*self*, *n1*, *n2*)

Insert a new edge into the graph.

Parameters

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

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

addNode(*self*)

Returns the new NodeId

Return type int

addNodeWithId(*self*, *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(*self*, *n*)

Add n nodes.

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

Returns the new ids

Return type Set of int

adjacents(*self*, *id*)

arcs(*self*)

Returns the list of the arcs

Return type List

children(*self*, *id*)

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

Returns the set of all the children

Return type Set

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

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

edges(*self*)

Returns the list of the edges

Return type List

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

Returns True if the graph is empty

Return type bool

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

Returns True if the graph doesn't contains arcs

Return type bool

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

Returns True if the graph doesn't contains edges

Return type bool

eraseArc(*self*, *n1*, *n2*)
Erase the arc between n1 and n2.

Parameters

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

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

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

eraseEdge(*self*, *n1*, *n2*)
Erase the edge between n1 and n2.

Parameters

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

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

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

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

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

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

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

existsArc(*self*, *n1*, *n2*)
Check if an arc exists bewteen n1 and n2.

Parameters

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

Returns True if the arc exists

Return type bool

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

Parameters

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

Returns True if the arc exists

Return type bool

existsNode(*self*, *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(*self*, *_from*, *to*)
Check if a directedpath exists bewteen from and to.

Parameters

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

Returns True if the directed path exists

Return type bool

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

Returns True if the graph contains a cycle

Return type bool

mixedOrientedPath(*self*, *node1*, *node2*)

Parameters

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

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

Return type List

mixedUnorientedPath(*self*, *node1*, *node2*)

Parameters

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

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

Return type List

neighbours(*self*, *id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*self*)

Returns the set of ids

Return type set

nodes2ConnectedComponent(*self*)

parents(*self*, *id*)

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

Returns the set of the parents ids.

Return type Set

partialUndiGraph(*self*, *nodes*)

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

Returns The partial graph formed by the nodes given in parameter

Return type *pyAgrum.UndiGraph* (page 9)

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

sizeEdges(*self*)

Returns the number of edges in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

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

Returns the list of the nodes Ids in a topological order

Return type List

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

CHAPTER 2

Random Variables

aGrUM/pyAgrum is currently dedicated for discrete probability distributions.

There are 3 types of discrete random variables in aGrUM/pyAgrum: LabelizedVariable, DiscretizedVariable and RangeVariable. The 3 types are mainly provided in order to ease modelization. Derived from DiscreteVariable, they share a common API. They essentially differ by the means to create, name and access to their modalities.

2.1 Common API for Random Discrete Variables

class pyAgrum.**DiscreteVariable**(**args*, ***kwargs*)

DiscreteVariable is the base class for discrete random variables.

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

Parameters:

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

DiscreteVariable(aDRV) -> DiscreteVariable

Parameters:

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

description(*self*)

Returns the description of the variable

Return type str

domain(*self*)

Returns the domain of the variable

Return type str

domainSize(*self*)

Returns the number of modalities in the variable domain

Return type int

empty(*self*)

Returns True if the domain size < 2

Return type bool

index(*self*, *label*)

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

Returns the indice of the label

Return type int

label(*self*, *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(*self*)

Returns a tuple containing the labels

Return type tuple

name(*self*)

Returns the name of the variable

Return type str

numerical(*self*, *indice*)

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

Returns the numerical representation of the indice-th value

Return type float

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

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

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

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

stype(*self*)

toDiscretizedVar(*self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toIntegerVar(*self*)

toLabelizedVar(*self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar(*self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

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

toStringWithDescription(*self*)

Returns a description of the variable

Return type str

varType(*self*)
returns the type of variable

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

Return type int

2.2 Concrete classes for Random Discrete Variables

2.2.1 LabelizedVariable

class pyAgrum.**LabelizedVariable**(**args*)
LabelizedVariable is a discrete random variable with a customizable sequence of labels.

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

Parameters:

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

LabelizedVariable(aLDRV) -> LabelizedVariable

Parameters:

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

Examples

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

(continues on next page)

(continued from previous page)

```
>>>
>>> vb.isLabel('E')
>>> ## False
>>>
>>> vb.label(2)
>>> ## 'B'
```

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

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

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

changeLabel (*self*, *pos*, *aLabel*)
Change the label at the specified index

Parameters

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

Raises

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

description (*self*)

Returns the description of the variable

Return type str

domain (*self*)

Returns the domain of the variable as a string

Return type str

domainSize (*self*)

Returns the number of modalities in the variable domain

Return type int

empty (*self*)

Returns True if the domain size < 2

Return type bool

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

index (*self*, *label*)

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

Returns the indice of the label

Return type int

isLabel (*self*, *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(*self*, *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(*self*)

Returns a tuple containing the labels

Return type tuple

name(*self*)

Returns the name of the variable

Return type str

numerical(*self*, *index*)

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

Returns the numerical representation of the indice-th value

Return type float

posLabel(*self*, *label*)

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

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

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

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

stype(*self*)

toDiscretizedVar(*self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toIntegerVar(*self*)

toLabelizedVar(*self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar(*self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

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

toStringWithDescription(*self*)

Returns a description of the variable

Return type str

varType(*self*)
returns the type of variable

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

Return type int

2.2.2 DiscretizedVariable

class pyAgrum.**DiscretizedVariable**(**args*)
DiscretizedVariable is a discrete random variable with a set of `ticks` defining intervalls.

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

Parameters:

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

DiscretizedVariable(aDDRV) -> DiscretizedVariable

Parameters:

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

Examples

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

addTick(**args*)

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

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

description(*self*)

Returns the description of the variable

Return type str

domain(*self*)

Returns the domain of the variable as a string

Return type str

domainSize(*self*)

Returns the number of modalities in the variable domain

Return type int

empty(*self*)

Returns True if the domain size < 2

Return type bool

eraseTicks(*self*)
erase all the Ticks

index(*self*, *label*)

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

Returns the indice of the label

Return type int

isTick(*self*, *aTick*)

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

Returns True if the Tick already exists

Return type bool

label(*self*, *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(*self*)

Returns a tuple containing the labels

Return type tuple

name(*self*)

Returns the name of the variable

Return type str

numerical(*self*, *indice*)

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

Returns the numerical representation of the indice-th value

Return type float

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

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

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

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

stype(*self*)

tick(*self*, *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(*self*)

Returns a tuple containing all the Ticks

Return type tuple

toDiscretizedVar(*self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toIntegerVar(*self*)

toLabelizedVar(*self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar(*self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

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

toStringWithDescription(*self*)

Returns a description of the variable

Return type str

varType(*self*)
returns the type of variable

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

Return type int

2.2.3 RangeVariable

class pyAgrum.**RangeVariable**(**args*)
RangeVariable represents a variable with a range of integers as domain.

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

Parameters:

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

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

Parameters:

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

By default `minVal=0` and `maxVal=1`

RangeVariable(aRV) -> RangeVariable

Parameters:

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

Examples

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

belongs(*self*, *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(*self*)

Returns the description of the variable

Return type str

domain(*self*)

Returns the domain of the variable

Return type str

domainSize(*self*)

Returns the number of modalities in the variable domain

Return type int

empty(*self*)

Returns True if the domain size < 2

Return type bool

index(*self*, *arg2*)

Parameters **arg2** (*str*) – a label

Returns the indice of the label

Return type int

label(*self*, *index*)

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(*self*)

Returns a tuple containing the labels

Return type tuple

maxVal(*self*)

Returns the upper bound of the variable.

Return type long

minVal(*self*)

Returns the lower bound of the variable

Return type long

name(*self*)

Returns the name of the variable

Return type str

numerical(*self*, *index*)

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

Returns the numerical representation of the indice-th value

Return type float

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

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

setMaxVal(*self*, *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(*self*, *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(*self*, *theValue*)
sets the name of the variable.

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

stype(*self*)

toDiscretizedVar(*self*)

Returns the discretized variable

Return type *pyAgrum.DiscretizedVariable* (page 26)

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

toIntegerVar(*self*)

toLabelizedVar(*self*)

Returns the labelized variable

Return type *pyAgrum.LabelizedVariable* (page 23)

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

toRangeVar(*self*)

Returns the range variable

Return type *pyAgrum.RangeVariable* (page 28)

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

toStringWithDescription(*self*)

Returns a description of the variable

Return type str

varType(*self*)
returns the type of variable

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

Return type int

CHAPTER 3

Potential and Instantiation

pyAgrum.Potential (page 39) is a multi-dimensional array with a *pyAgrum.DiscreteVariable* (page 21) associated to each dimension. It is used to represent probabilities and utilities tables in aGrUMs' multidimensional (graphical) models with some conventions.

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

```
>>> a=gum.RangeVariable("A","variable A",1,3)
>>> b=gum.RangeVariable("B","variable B",1,2)
>>> p=gum.Potential().add(a).add(b).fillWith([1,2,3,4,5,6]);
>>> print(p)
<A:1|B:1> :: 1 /<A:2|B:1> :: 2 /<A:3|B:1> :: 3 /<A:1|B:2> :: 4 /<A:2|B:2> :: 5 /
↪<A:3|B:2> :: 6
```

- If a *pyAgrum.Potential* (page 39) with the sequence of *pyAgrum.DiscreteVariable* (page 21) X,Y,Z represents a conditional probability Table (CPT), it will be P(X|Y,Z).

```
>>> print(p.normalizeAsCPT())
<A:1|B:1> :: 0.166667 /<A:2|B:1> :: 0.333333 /<A:3|B:1> :: 0.5 /<A:1|B:2> :: 0.
↪266667 /<A:2|B:2> :: 0.333333 /<A:3|B:2> :: 0.4
```

- For addressing and looping in a *pyAgrum.Potential* (page 39) structure, pyAgrum provides `Instantiation` class which represents a multi-dimensionnal index.

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

- *pyAgrum.Potential* (page 39) include tensor operators (see for instance this notebook (http://www-desir.lip6.fr/~phw/aGrUM/docs/last/notebooks/potentials.ipynb.html)).

```
>>> c=gum.RangeVariable("C","variable C",1,5)
>>> q=gum.Potential().add(a).add(c).fillWith(1)
>>> print(p+q)
<A:1|C:1|B:1> :: 2 /<A:2|C:1|B:1> :: 3 /<A:3|C:1|B:1> :: 4 /<A:1|C:2|B:1> :: 2 /
↪<A:2|C:2|B:1> :: 3 /<A:3|C:2|B:1> :: 4 /<A:1|C:3|B:1> :: 2 /<A:2|C:3|B:1> :: 3 /
↪<A:3|C:3|B:1> :: 4 /<A:1|C:4|B:1> :: 2 /<A:2|C:4|B:1> :: 3 /<A:3|C:4|B:1> :: 4 /
↪<A:1|C:5|B:1> :: 2 /<A:2|C:5|B:1> :: 3 /<A:3|C:5|B:1> :: 4 /<A:1|C:1|B:2> :: 5 /
↪<A:2|C:1|B:2> :: 6 /<A:3|C:1|B:2> :: 7 /<A:1|C:2|B:2> :: 5 /<A:2|C:2|B:2> :: 6 /
↪<A:3|C:2|B:2> :: 7 /<A:1|C:3|B:2> :: 5 /<A:2|C:3|B:2> :: 6 /<A:3|C:3|B:2> :: 7 /
↪<A:1|C:4|B:2> :: 5 /<A:2|C:4|B:2> :: 6 /<A:3|C:4|B:2> :: 7 /<A:1|C:5|B:2> :: 5 /
↪<A:2|C:5|B:2> :: 6 /<A:3|C:5|B:2> :: 7
>>> print((p*q).margSumOut(["B","C"])) # marginalize p*q over B and C(using sum)
<A:1> :: 25 /<A:2> :: 35 /<A:3> :: 45
```

3.1 Instantiation

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

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

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

Instantiation() -> Instantiation default constructor

Instantiation(aI) -> Instantiation

Parameters:

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

Returns

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

Examples

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

add(*self*, *v*)

Adds a new variable in the Instantiation.

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

Raises *DuplicateElement* (page 249) – If the variable is already in this Instantiation

addVarsFromModel(*model*, *names*)

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

Parameters

- **model** (*pyAgrum.GraphicalModel*) –
- **(discrete) graphical model such as Bayesian network, Markov network, Influence Diagram, etc.** (*a*) –
- **names** (*iterable of strings*) –

- **list/set/etc of names of variables (as string)** (*a*) –

Returns

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

chgVal (*self*, *v*, *newval*)
chgVal(self, v, newval) -> Instantiation chgVal(self, varPos, newval) -> Instantiation chgVal(self, var, newval) -> Instantiation chgVal(self, var, newval) -> Instantiation

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

Parameters

- **v** (pyAgrum.DiscreteVariable (page 21) *or string*) – The variable whose value is assigned (or its name)
- **varPos** (*int*) – The index of the variable whose value is assigned in the tuple of variables of the Instantiation
- **newval** (*int or string*) – The index of the value assigned (or its name)

Returns The modified instantiation

Return type *pyAgrum.Instantiation* (page 34)

Raises

- *NotFound* (page 255) – If variable v does not belong to the instantiation.
- OutOfBound – If newval is not a possible value for the variable.

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

contains (*self*, *v*)
contains(self, name) -> bool contains(self, v) -> bool

Indicates whether a given variable belongs to the Instantiation.

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

Returns True if the variable is in the Instantiation.

Return type bool

dec (*self*)
Operator –.

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

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

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

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

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

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

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

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

Raises *NotFound* (page 255) – If variable v does not belong to the Instantiation.

domainSize(*self*)

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

Return type int

empty(*self*)

Returns True if the instantiation is empty.

Return type bool

end(*self*)

Returns True if the Instantiation reached the end.

Return type bool

erase(*self*, *v*)
erase(self, name)

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

Raises *NotFound* (page 255) – If v does not belong to this Instantiation.

fromdict(*self*, *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(*self*)

Returns the hamming distance of this instantiation.

Return type int

inOverflow(*self*)

Returns True if the current value of the tuple is correct

Return type bool

inc(*self*)
Operator ++.

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

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

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

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

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

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

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

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

Raises *NotFound* (page 255) – If variable v does not belong to the Instantiation.

isMutable(*self*)

nbrDim(*self*)

Returns The number of variables in the Instantiation.

Return type int

pos(*self*, *v*)

Returns the position of the variable v.

Return type int

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

Raises *NotFound* (page 255) – If v does not belong to the instantiation.

rend(*self*)

Returns True if the Instantiation reached the rend.

Return type bool

reorder(*self*, *v*)
reorder(self, i)

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

Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

setMutable (*self*)

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

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

Returns a reference to the instantiation

Return type *pyAgrum.Instantiation* (page 34)

todict (*self*, *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 (*self*)
Alias for unsetOverflow().

unsetOverflow (*self*)
Removes the flag overflow.

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

Parameters

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

Returns the current value of the variable.

Return type int

Raises *NotFound* (page 255) – If the element cannot be found.

variable(*self*, *i*)
variable(self, name) -> DiscreteVariable

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

Returns the variable at position i in the tuple.

Return type *pyAgrum.DiscreteVariable* (page 21)

Raises *NotFound* (page 255) – If the element cannot be found.

variablesSequence(*self*)

Returns the sequence of DiscreteVariable of this instantiation.

Return type List

3.2 Potential

class pyAgrum.**Potential**(**args*)
Class representing a potential.

Potential() -> Potential default constructor

Potential(src) -> Potential

Parameters:

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

KL(*self*, *p*)
Check the compatibility and compute the Kullback-Leibler divergence between the potential and.

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

Returns The value of the divergence

Return type float

Raises

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

abs(*self*)
Apply abs on every element of the container

Returns a reference to the modified potential.

Return type *pyAgrum.Potential* (page 39)

add(*self*, *v*)
Add a discrete variable to the potential.

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

Raises

- *DuplicateElement* (page 249) – If the variable is already in this Potential.
- *InvalidArgument* (page 252) – If the variable is empty.

argmax(*self*)

argmin(*self*)

contains(*self*, *v*)

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

Returns True if the var is in the potential

Return type bool

domainSize(*self*)

draw(*self*)
draw a value using the potential as a probability table.

Returns the index of the drawn value

Return type int

empty(*self*)

Returns Returns true if no variable is in the potential.

Return type bool

entropy(*self*)

Returns the entropy of the potential

Return type double

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

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

Parameters

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

Returns the new Potential

Return type *pyAgrum.Potential* (page 39)

fillWith(*self*, *src*)
fillWith(self, src, mapSrc) -> Potential fillWith(self, data) -> Potential fillWith(self, val) -> Potential

Automatically fills the potential with v.

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

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

Returns a reference to the modified potentia

Return type *pyAgrum.Potential* (page 39)

Raises gum.SizeError – If v size's does not matches the domain size.

fillWithFunction(*s*, *noise=None*)
Automatically fills the potential as a (quasi) deterministic CPT with the evaluation of the expression s.

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

Examples

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

Parameters

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

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

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

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

findAll(*self*, *v*)

get(*self*, *i*)

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

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

Return type double

inverse(*self*)

isNonZeroMap(*self*)

Returns a boolean-like potential using the predicate isNonZero

Return type *pyAgrum.Potential* (page 39)

log2(*self*)
log2 all the values in the Potential

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

loopIn()
Generator to iterate inside a Potential.

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

Examples

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

(continues on next page)

(continued from previous page)

```
    print(bn.cpt("B").get(i))
    bn.cpt("B").set(i,0.3)
```

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

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

Returns the projected Potential

Return type *pyAgrum.Potential* (page 39)

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

max(*self*)

Returns the maximum of all elements in the Potential

Return type double

maxNonOne(*self*)

Returns the maximum of non one elements in the Potential

Return type double

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

min(*self*)

Returns the min of all elements in the Potential

Return type double

minNonZero(*self*)

Returns the min of non zero elements in the Potential

Return type double

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

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

Returns the number of vars in the multidimensional container.

Return type int

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

Returns a reference to the new Potential

Return type *pyAgrum.Potential* (page 39)

new_abs(*self*)

new_log2(*self*)

new_sq(*self*)

noising(*self*, *alpha*)

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

Returns a reference to the normalized Potential

Return type *pyAgrum.Potential* (page 39)

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

Returns a reference to the normalized Potential

Return type *pyAgrum.Potential* (page 39)

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

pos (*self*, *v*)

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

Returns

Return type Returns the index of a variable.

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

product (*self*)

Returns the product of all elements in the Potential

Return type double

putFirst (*self*, *varname*)

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

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

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

random (*self*)

randomCPT (*self*)

randomDistribution (*self*)

remove (*self*, *var*)

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

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

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

reorganize (*self*, *vars*)
reorganize(self, vars) -> Potential

Create a new Potential with another order.

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

Return type list

Returns a reference to the modified potential

Return type *pyAgrum.Potential* (page 39)

scale (*self*, *v*)
Create a new potential multiplied by v.

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

Returns

Return type a reference to the modified potential

set(*self*, *i*, *value*)
Change the value pointed by i

Parameters

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

sq(*self*)
Square all the values in the Potential

sum(*self*)

Returns the sum of all elements in the Potential

Return type double

toarray()

Returns the potential as an array

Return type array

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

tolatex()
Render object to a LaTeX tabular.

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

Returns the potential as LaTeX string

Return type str

tolist()

Returns the potential as a list

Return type list

topandas()

Returns the potential as an pandas.DataFrame

Return type pandas.DataFrame

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

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

Returns

Return type a reference to the modified potential

var_dims

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

Return type list

var_names

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

Return type list

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

variable(*self*, *i*)

variable(self, name) -> DiscreteVariable

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

Returns

Return type the varible at the ith index

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

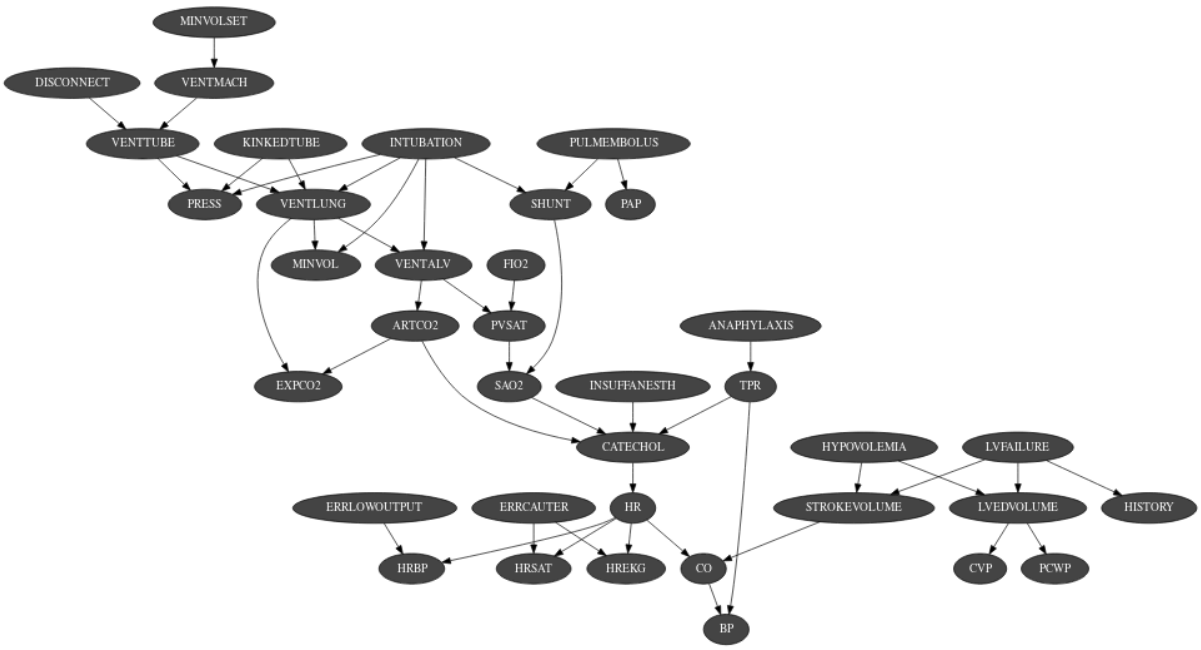
variablesSequence()

Returns a list containing the sequence of variables

Return type list

CHAPTER 4

Bayesian network



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

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

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

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

Tutorial

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

Reference

4.1 Model

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

BayesNet represents a Bayesian network.

BayesNet(name='') -> BayesNet

Parameters:

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

BayesNet(source) -> BayesNet

Parameters:

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

add(*self*, *var*)

add(self, name, nbrmod) -> int add(self, var, aContent) -> int add(self, var, id) -> int add(self, var, aContent, id) -> int

Add a variable to the pyAgrum.BayesNet.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable added
- **name** (*str*) – the variable name
- **nbrmod** (*int*) – the number of modalities for the new variable
- **id** (*int*) – the variable forced id in the pyAgrum.BayesNet

Returns the id of the new node

Return type int

Raises

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

addAMPLITUDE(*self*, *var*)

Others aggregators

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

Returns the id of the added value

Return type int

addAND(*self*, *var*)

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

The id of the new variable is automatically generated.

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – The variable added by copy.

Returns the id of the added variable.

Return type int

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

addArc(*self*, *tail*, *head*)
addArc(self, tail, 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(*self*, *var*, *value=1*)
Others aggregators

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

Returns the id of the added value

Return type int

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

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

Returns the id of the added value

Return type int

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

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

Returns the id of the added variable.

Return type int

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

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

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

Parameters

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

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id is already used

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

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

Returns the id of the added value

Return type int

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

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

Returns the id of the added value

Return type int

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

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

Returns the id of the added value

Return type int

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

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

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

Parameters

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

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id is already used

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

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

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

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

Parameters

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

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id is already used

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

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

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

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

Parameters

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

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id is already used

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

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

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

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

Parameters

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

Returns the id of the added variable.

Return type int

addOR(*self*, *var*)
Add a variable, it's associate node and an OR implementation.

The id of the new variable is automatically generated.

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

Parameters **variable** (pyAgrum.DiscreteVariable (page 21)) – The variable added by copy

Returns the id of the added variable.

Return type int

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

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

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

Returns the id of the added value

Return type int

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

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

Parameters

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

addWeightedArc(*self*, *tail*, *head*, *causalWeight*)

addWeightedArc(self, tail, head, causalWeight)

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

Parameters

- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)
- **causalWeight** (*double*) – the added causal weight

Raises

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

ancestors(*self*, *norid*)

arcs(*self*)

Returns The lisf of arcs in the IBayesNet

Return type list

beginTopologyTransformation(*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(*self*, *id*, *newPot*)

changePotential(self, name, newPot)

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

Parameters

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

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

changeVariableLabel(*self*, *id*, *old_label*, *new_label*)
changeVariableLabel(self, name, old_label, 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(*self*, *id*, *new_name*)
changeVariableName(self, name, 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(*self*, *norid*)

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

Returns the set of all the children

Return type Set

clear(*self*)
Clear the whole BayesNet

completeInstantiation(*self*)

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

cpt(*self*, *varId*)
cpt(self, name) -> Potential

Returns the CPT of a variable.

Parameters

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

Returns The variable's CPT.

Return type *pyAgrum.Potential* (page 39)

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

dag(*self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

descendants(*self*, *norid*)

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

Returns the dimension of the BayesNet

Return type int

empty(*self*)

endTopologyTransformation(*self*)
Terminates a sequence of insertions/deletions of arcs by adjusting all CPTs dimensions. End Multiple Change for all CPTs.

Returns

Return type *pyAgrum.BayesNet* (page 48)

erase(*self*, *varId*)
erase(self, name) erase(self, var)

Remove a variable from the pyAgrum.BayesNet.

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

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

Parameters

- **id** (*int*) – The variable's id to remove.
- **name** (*str*) – The variable's name to remove.
- **var** (`pyAgrum.DiscreteVariable` (page 21)) – A reference on the variable to remove.

eraseArc(*self*, *arc*)
eraseArc(self, tail, head) eraseArc(self, tail, head)

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

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

Parameters

- **arc** (`pyAgrum.Arc` (page 3)) – The arc to be removed.
- **head** – a variable's id (int)
- **tail** – a variable's id (int)
- **head** – a variable's name (str)
- **tail** – a variable's name (str)

exists(*self*, *node*)

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

family(*self*, *norid*)

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

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

- the structure 'a->b->c;b->d<-e;'.
- the type of the variables with different syntax:
  - by default, a variable is a gum.RangeVariable using the default domain size (second argument)
  - with 'a[10]', the variable is a gum.RangeVariable using 10 as domain size (from 0 to 9)
  - with 'a[3,7]', the variable is a gum.RangeVariable using a domainSize from 3 to 7
  - with 'a[1,3.14,5,6.2]', the variable is a gum.DiscretizedVariable using the given ticks (at least 3 values)
  - with 'a{top|middle|bottom}', the variable is a gum.LabelizedVariable using the given labels.
  - with 'a{-1|5|0|3}', the variable is a gum.IntegerVariable using the sorted given values.

Note:

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

Examples

```
>>> import pyAgrum as gum
>>> bn=gum.BayesNet.fastPrototype('A->B[1,3]<-C{yes|No}->D[2,4]<-E[1,2.5,3.
↪9]',6)
```

Parameters

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

Returns the resulting Bayesian network

Return type *pyAgrum.BayesNet* (page 48)

generateCPT(*self*, *node*)
generateCPT(self, 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(*self*)
Randomly generates CPTs for a given structure.

hasSameStructure(*self*, *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(*self*, *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(*self*, *names*)

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

jointProbability(*self*, *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(*self*, *name*, *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(*self*, *name*, *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(*self*, *name*, *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(*self*, *name*, *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(*self*, *name*, *system=""*, *classpath=""*, *l=(PyObject *) 0*)
Load an O3PRM file.

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

Parameters

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

Raises

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

loadUAI(*self*, *name*, *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(*self*)

log2JointProbability(*self*, *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(*self*)

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

Return type double

maxParam(*self*)

Returns the biggest value in the CPTs of the BayesNet

Return type double

maxVarDomainSize(*self*)

Returns the biggest domain size among the variables of the BayesNet

Return type int

minNonZeroParam(*self*)

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

Return type double

minParam(*self*)

Returns the smallest value in the CPTs of the IBayesNet

Return type double

minimalCondSet(*self*, *target*, *list*)

minimalCondSet(self, targets, 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(*self*, *clear=True*)

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

Returns The moral graph

Return type *pyAgrum.UndiGraph* (page 9)

moralizedAncestralGraph(*self*, *nodes*)

names(*self*)

Returns The names of the graph variables

Return type list

nodeId(*self*, *var*)

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

Returns the id of the variable

Return type int

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

nodes(*self*)

Returns the set of ids

Return type set

nodeset(*self*, *names*)

parents(*self*, *norid*)

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

Returns the set of the parents ids.

Return type Set

property(*self*, *name*)

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

reverseArc(*self*, *tail*, *head*)
reverseArc(self, tail, head) reverseArc(self, arc)

Reverses an arc while preserving the same joint distribution.

Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs (*self*)

Returns the number of arcs in the graph

Return type int

toDot (*self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder (*self*, *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 (*self*, *id*)
variable(self, name) -> DiscreteVariable

Parameters

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

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

variableFromName (*self*, *name*)

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

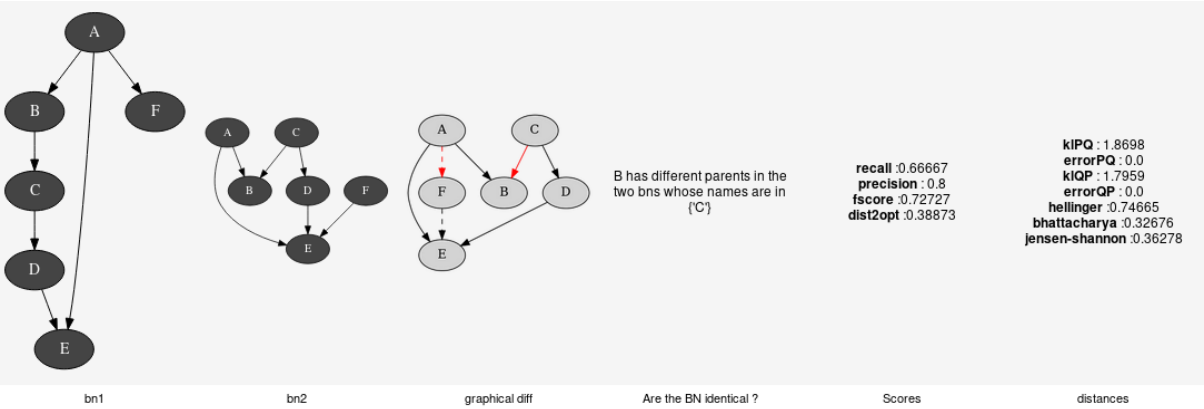
Raises `gum.IndexError` – If the graph does not contain the variable

variableNodeMap (*self*)

Returns the variable node map

Return type pyAgrum.variableNodeMap

4.2 Tools for Bayesian networks



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

4.2.1 Generation of database

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

BNDatabaseGenerator is used to easily generate databases from a gum.BayesNet.

BNDatabaseGenerator(bn) -> BNDatabaseGenerator

Parameters:

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

database(*self*)

drawSamples(*self*, *nbSamples*)

log2likelihood(*self*)

setAntiTopologicalVarOrder(*self*)

setRandomVarOrder(*self*)

setTopologicalVarOrder(*self*)

setVarOrder(*self*, *varOrder*)
setVarOrder(self, varOrder)

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

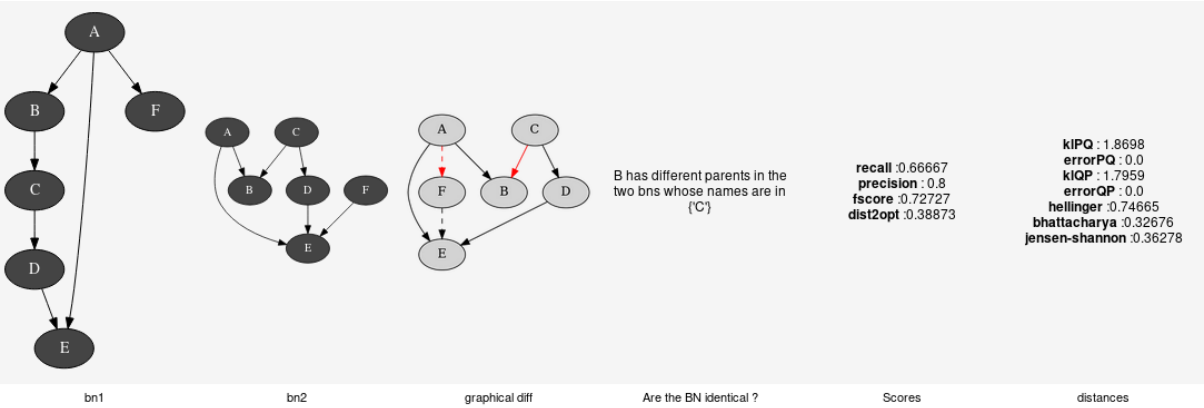
toCSV(*self*, *csvFileURL*, *useLabels=True*, *append=False*, *csvSeparator=','*, *checkOnAppend=False*)

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

varOrder(*self*)

varOrderNames(*self*)

4.2.2 Comparison of Bayesian networks



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

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

Class representing exacte computation of divergence and distance between BNs

ExactBNdistance(P,Q) -> ExactBNdistance

Parameters:

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

ExactBNdistance(ebnd) -> ExactBNdistance

Parameters:

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

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

compute(*self*)

Returns a dictionnary containing the different values after the computation.

Return type dict

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

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

GibbsBNdistance(P,Q) -> GibbsBNdistance

Parameters:

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

GibbsBNdistance(gbnd) -> GibbsBNdistance

Parameters:

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

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

burnIn(*self*)

Returns size of burn in on number of iteration

Return type int

compute(*self*)

Returns a dictionnary containing the different values after the computation.

Return type dict

continueApproximationScheme(*self*, *error*)

Continue the approximation scheme.

Parameters **error** (*double*) –

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

disableEpsilon(*self*)

Disable epsilon as a stopping criterion.

disableMaxIter(*self*)

Disable max iterations as a stopping criterion.

disableMaxTime(*self*)

Disable max time as a stopping criterion.

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

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

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

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

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

epsilon(*self*)

Returns the value of epsilon

Return type double

history(*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(*self*)
Initiate the approximation scheme.

isDrawnAtRandom(*self*)

Returns True if variables are drawn at random

Return type bool

isEnabledEpsilon(*self*)

Returns True if epsilon is used as a stopping criterion.

Return type bool

isEnabledMaxIter(*self*)

Returns True if max iterations is used as a stopping criterion

Return type bool

isEnabledMaxTime(*self*)

Returns True if max time is used as a stopping criterion

Return type bool

isEnabledMinEpsilonRate(*self*)

Returns True if epsilon rate is used as a stopping criterion

Return type bool

maxIter(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*self*)

Returns the number of variable drawn at each iteration

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

remainingBurnIn(*self*)

Returns the number of remaining burn in

Return type int

setBurnIn(*self*, *b*)

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

setDrawnAtRandom(*self*, *_atRandom*)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – If eps<0

setMaxIter(*self*, *max*)

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

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

setMaxTime(*self*, *timeout*)

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

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

setMinEpsilonRate(*self*, *rate*)

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

setNbrDrawnVar(*self*, *_nbr*)

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

setPeriodSize(*self*, *p*)

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

Raises `gum.OutOfBounds` – If p<1

setVerbosity(*self*, *v*)

Parameters **v** (*bool*) – verbosity

startOfPeriod(*self*)

Returns True if it is a start of a period

Return type bool

stateApproximationScheme(*self*)

Returns the state of the approximation scheme

Return type int

stopApproximationScheme(*self*)
Stop the approximation scheme.

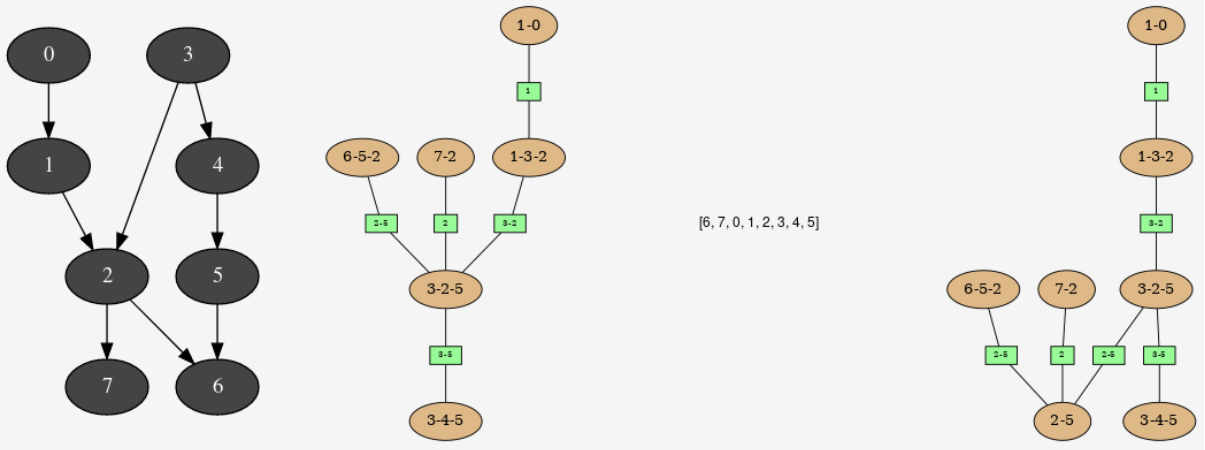
updateApproximationScheme(*self*, *incr=1*)
Update the approximation scheme.

verbosity(*self*)

Returns True if the verbosity is enabled

Return type bool

4.2.3 Explanation and analysis



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

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

JunctionTreeGenerator() -> JunctionTreeGenerator default constructor

binaryJoinTree(*self*, *g*, *partial_order=None*)
binaryJoinTree(self, dag, partial_order=None) -> CliqueGraph binaryJoinTree(self, bn, partial_order=None) -> CliqueGraph

Computes the binary joint tree for its parameters. If the first parameter is a graph, the heurisitcs assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

Parameters

- **g** (pyAgrum.UndiGraph (page 9)) – a undirected graph

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

Returns the current binary joint tree

Return type *pyAgrum.CliqueGraph* (page 12)

eliminationOrder(*self*, *g*, *partial_order=None*)

eliminationOrder(self, dag, partial_order=None) -> PyObject eliminationOrder(self, bn, partial_order=None) -> PyObject

Computes the elimination for its parameters. If the first parameter is a graph, the heurisitcs assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

Parameters

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

Returns the current elimination order.

Return type *pyAgrum.CliqueGraph* (page 12)

junctionTree(*self*, *g*, *partial_order=None*)

junctionTree(self, dag, partial_order=None) -> CliqueGraph junctionTree(self, bn, partial_order=None) -> CliqueGraph junctionTree(self, mn, partial_order=None) -> CliqueGraph

Computes the junction tree for its parameters. If the first parameter is a graph, the heurisitcs assume that all the node have the same domain size (2). If given, the heuristic takes into account the partial order for its elimination order.

Parameters

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

Returns the current junction tree.

Return type *pyAgrum.CliqueGraph* (page 12)

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

Proxy of C++ pyAgrum.EssentialGraph class.

arcs(*self*)

Returns The lisf of arcs in the EssentialGraph

Return type list

children(*self*, *id*)

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

Returns the set of all the children

Return type Set

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

edges(*self*)

Returns the list of the edges

Return type List

mixedGraph(*self*)

Returns the mixed graph

Return type *pyAgrum.MixedGraph* (page 16)

neighbours(*self*, *id*)

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

Returns The set of edges adjacent to the given node

Return type Set

nodes(*self*)

parents(*self*, *id*)

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

Returns the set of the parents ids.

Return type Set

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

sizeEdges(*self*)

Returns the number of edges in the graph

Return type int

sizeNodes(*self*)

Returns the number of nodes in the graph

Return type int

skeleton(*self*)

toDot(*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(*self*)

Returns the list of the arcs

Return type List

children(*self*, *id*)

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

Returns the set of all the children

Return type Set

connectedComponents()

connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

dag(*self*)

Returns a copy of the DAG

Return type *pyAgrum.DAG* (page 7)

hasSameStructure(*self*, *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(*self*)

Returns the set of ids

Return type set

parents(*self*, *id*)

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

Returns the set of the parents ids.

Return type Set

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

sizeNodes(*self*)

Returns the number of nodes in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

4.2.4 Fragment of Bayesian networks

This class proposes a shallow copy of a part of Bayesian network. It can be used as a Bayesian network for inference algorithms (for instance).

class pyAgrum.**BayesNetFragment**(*bn: pyAgrum.IBayesNet*)

BayesNetFragment represents a part of a Bayesian network (subset of nodes). By default, the arcs and the CPTs are the same as the BN but local CPTs can be build to express different local dependencies. All the non local CPTs are not copied. Therefore a BayesNetFragment is a light object.

BayesNetFragment(BayesNet bn) -> BayesNetFragment

Parameters:

- **bn** (*pyAgrum.BayesNet*) – the bn refered by the fragment

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

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

Parameters

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

ancestors(*self*, *norid*)

arcs(*self*)

Returns The lisf of arcs in the IBayesNet

Return type list

checkConsistency(*self*, *id*)

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

If a variable is added to the fragment but not its parents, there is no CPT consistant for this variable. This function checks the consistency for a variable of for all.

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

Returns True if the variable(s) is consistant.

Return type boolean

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

children(*self*, *norid*)

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

Returns the set of all the children

Return type Set

completeInstantiation(*self*)

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

cpt(*self*, *varId*)
cpt(self, name) -> Potential

Returns the CPT of a variable.

Parameters

- **VarId** (*int*) – A variable's id in the pyAgrum.IBayesNet.
- **name** (*str*) – A variable's name in the pyAgrum.IBayesNet.

Returns The variable's CPT.

Return type *pyAgrum.Potential* (page 39)

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

dag(*self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

descendants(*self*, *norid*)

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

Returns the dimension of the BayesNet

Return type int

empty(*self*)

exists(*self*, *node*)

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

family(*self*, *norid*)

hasSameStructure(*self*, *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(*self*, *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(*self*, *names*)

installAscendants(*self*, *id*)
installAscendants(self, name)

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

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

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

installCPT(*self*, *id*, *pot*)
installCPT(self, name, pot)

Install a local CPT for a node. Doing so, it changes the parents of the node in the fragment.

Parameters

- **n** (*int, str*) – the id or the name of the variable.
- **pot** (Potential (page 39)) – the Potential to install

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

installMarginal(*self*, *id*, *pot*)
installMarginal(self, name, pot)

Install a local marginal for a node. Doing so, it removes the parents of the node in the fragment.

Parameters

- **n** (*int, str*) – the id or the name of the variable.
- **pot** (Potential (page 39)) – the Potential (marginal) to install

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

installNode(*self*, *id*)
installNode(self, name)

Add a node to the fragment. The arcs that can be added between installed nodes are created. No specific CPT are created. Then either the parents of the node are already in the fragment and the node is consistant, or the parents are not in the fragment and the node is not consistant.

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

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

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

isInstalledNode(*self*, *id*)
isInstalledNode(self, name) -> bool

Check if a node is in the fragment

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

jointProbability(*self*, *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(*self*)

log2JointProbability(*self*, *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(*self*)

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

Return type double

maxParam(*self*)

Returns the biggest value in the CPTs of the BayesNet

Return type double

maxVarDomainSize(*self*)

Returns the biggest domain size among the variables of the BayesNet

Return type int

minNonZeroParam(*self*)

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

Return type double

minParam(*self*)

Returns the smallest value in the CPTs of the IBayesNet

Return type double

minimalCondSet(*self*, *target*, *list*)
minimalCondSet(self, targets, 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(*self*, *clear=True*)
Returns the moral graph of the BayesNet, formed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected.

Returns The moral graph

Return type *pyAgrum.UndiGraph* (page 9)

moralizedAncestralGraph(*self*, *nodes*)

names(*self*)

Returns The names of the graph variables

Return type list

nodeId(*self*, *var*)

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

Returns the id of the variable

Return type int

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

nodes(*self*)

Returns the set of ids

Return type set

nodeset(*self*, *names*)

parents(*self*, *norid*)

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

Returns the set of the parents ids.

Return type Set

property(*self*, *name*)

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

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

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

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

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

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*self*, *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(*self*, *id*)
uninstallCPT(self, name)

Remove a local CPT. The fragment can become inconsistant.

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

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

uninstallNode(*self*, *id*)
uninstallNode(self, name)

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

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

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

variable(*self*, *id*)
variable(self, name) -> DiscreteVariable

Parameters

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

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

variableFromName(*self*, *name*)

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

Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

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

variableNodeMap(*self*)

Returns the variable node map

Return type pyAgrum.variableNodeMap

whenArcAdded(*self*, *src*, *_from*, *to*)

whenArcDeleted(*self*, *src*, *_from*, *to*)

whenNodeAdded(*self*, *src*, *id*)

whenNodeDeleted(*self*, *src*, *id*)

4.3 Inference

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

4.4 Exact Inference

4.4.1 Lazy Propagation

Lazy Propagation is the main exact inference for classical Bayesian networks in aGrUM/pyAgrum.

class pyAgrum.**LazyPropagation**(**args*)
Class used for Lazy Propagation

LazyPropagation(bn) -> LazyPropagation

Parameters:

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

BN(*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(*self*, *X*)
H(self, 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(*self*, *X*, *Y*)
I(self, X, Y) -> double

Parameters

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

Returns

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

VI(*self*, *X*, *Y*)
VI(self, X, Y) -> double

Parameters

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

Returns

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

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

addEvidence(*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*)
Removes all the evidence entered into the network.

eraseAllJointTargets(*self*)
Clear all previously defined joint targets.

eraseAllMarginalTargets(*self*)
Clear all the previously defined marginal targets.

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

evidenceJointImpact(*self*, *targets*, *evs*)
evidenceJointImpact(self, targets, evs) -> Potential

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

Parameters

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

Returns a Potential for P(target|evs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceProbability(*self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*self*, *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(*self*, *variable*)
isTarget(self, 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(*self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*self*, *targets*)

jointPosterior(*self*, *targets*)

Compute the joint posterior of a set of nodes.

Parameters **list** – the list of nodes whose posterior joint probability is wanted

> **Warning:** The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used bu the Potential.

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

Return type *pyAgrum.Potential* (page 39)

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

jointTargets(*self*)

Returns the list of target sets

Return type list

junctionTree(*self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*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(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

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

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

Raises

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

setFindBarrenNodesType(*self*, *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(*self*, *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(*self*, *new_triangulation*)

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*self*)

Returns the list of marginal targets

Return type list

updateEvidence(*evidces*)
Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

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

Raises

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

4.4.2 Shafer Shenoy Inference

class pyAgrum.**ShaferShenoyInference**(**args*)
Class used for Shafer-Shenoy inferences.

ShaferShenoyInference(bn) -> ShaferShenoyInference

Parameters:

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

BN(*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(*self*, *X*)
H(self, 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 (*self*, *X*, *Y*)
I(self, X, Y) -> double

Parameters

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

Returns

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

VI (*self*, *X*, *Y*)
VI(self, X, Y) -> double

Parameters

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

Returns

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

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

addEvidence (*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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 (*self*, *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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*)
Removes all the evidence entered into the network.

eraseAllJointTargets(*self*)
Clear all previously defined joint targets.

eraseAllMarginalTargets(*self*)
Clear all the previously defined marginal targets.

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

evidenceJointImpact(*self*, *targets*, *evs*)
evidenceJointImpact(self, targets, evs) -> Potential

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

Parameters

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

Returns a Potential for P(target|evs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceProbability(*self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*self*, *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(*self*, *variable*)
isTarget(self, 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(*self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*self*, *targets*)

jointPosterior(*self*, *targets*)
Compute the joint posterior of a set of nodes.

Parameters **list** – the list of nodes whose posterior joint probability is wanted

Warning: The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used bu the Potential.

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

Return type *pyAgrum.Potential* (page 39)

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

jointTargets(*self*)

Returns the list of target sets

Return type list

junctionTree(*self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*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(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

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

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

Raises

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

setFindBarrenNodesType(*self*, *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(*self*, *new_triangulation*)

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*self*)

Returns the list of marginal targets

Return type list

updateEvidence(*evidces*)
Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

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

Raises

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

4.4.3 Variable Elimination

class pyAgrum.**VariableElimination**(**args*)

Class used for Variable Elimination inference algorithm.

VariableElimination(bn) -> VariableElimination

Parameters:

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

BN(*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(*self*, *X*)

H(self, 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(*self*)

Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*)
Removes all the evidence entered into the network.

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

evidenceJointImpact(*self*, *targets*, *evs*)
Create a pyAgrum.Potential for P(joint targets|evs) (for all instanciation of targets and evs)

Parameters

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

Returns a Potential for P(target|evs)

Return type *pyAgrum.Potential* (page 39)

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

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*self*, *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(*self*, *variable*)
isTarget(self, 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(*self*, *targets*)

jointPosterior(*self*, *targets*)
Compute the joint posterior of a set of nodes.

Parameters **list** – the list of nodes whose posterior joint probability is wanted

Warning: The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used bu the Potential.

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

Return type *pyAgrum.Potential* (page 39)

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

jointTargets(*self*)

Returns the list of target sets

Return type list

junctionTree(*self*, *id*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*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(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

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

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

Raises

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

setFindBarrenNodesType(*self*, *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(*self*, *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(*self*, *new_triangulation*)

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*self*)

Returns the list of marginal targets

Return type list

updateEvidence(*evidces*)
Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

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

Raises

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

4.5 Approximated Inference

4.5.1 Loopy Belief Propagation

class pyAgrum.**LoopyBeliefPropagation**(*bn: pyAgrum.IBayesNet*)
Class used for inferences using loopy belief propagation algorithm.

LoopyBeliefPropagation(bn) -> LoopyBeliefPropagation

Parameters:

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

BN(*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(*self*, *X*)
H(self, 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(*self*)
Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*, *variable*)
isTarget(self, 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(*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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)

posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*self*)

Returns True if the verbosity is enabled

Return type bool

4.5.2 Sampling

Gibbs Sampling

class pyAgrum.**GibbsSampling**(*bn: pyAgrum.IBayesNet*)

Class for making Gibbs sampling inference in Bayesian networks.

GibbsSampling(bn) -> GibbsSampling

Parameters:

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

BN(*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(*self*, *X*)

H(self, 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(*self*)

Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)
addTarget(self, 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(*self*)

Returns size of burn in on number of iteration

Return type int

chgEvidence(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*)

Returns True if variables are drawn at random

Return type bool

isTarget(*self*, *variable*)
isTarget(self, 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(*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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*self*)

Returns the number of variable drawn at each iteration

Return type int

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)

posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setBurnIn(*self*, *b*)

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

setDrawnAtRandom(*self*, *_atRandom*)

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

setEpsilon(*self*, *eps*)

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

Raises gum.OutOfBounds – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setNbrDrawnVar(*self*, *_nbr*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)

H(self, 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(*self*)

Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)

addTarget(self, 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(*self*, *id*, *val*)

chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(targetlevs) (for all instanciation of target and evs)

Parameters

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

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

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

Returns a Potential for P(targetslevs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes (*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*self*, *id*)
hasEvidence(self, 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 (*self*, *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 (*self*, *id*)
hasSoftEvidence(self, 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 (*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 (*self*, *variable*)
isTarget(self, 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(*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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

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

setMaxTime(*self*, *timeout*)

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

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

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises `gum.OutOfBounds` – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)
H(self, 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(*self*)
Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)

addTarget(self, 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(*self*, *id*, *val*)

chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(targetlevs) (for all instanciation of target and evs)

Parameters

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

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

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

Returns a Potential for P(targetslevs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes (*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence (*self*, *id*)
hasEvidence(self, 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 (*self*, *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 (*self*, *id*)
hasSoftEvidence(self, 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 (*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 (*self*, *variable*)
isTarget(self, 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(*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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

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

setMaxTime(*self*, *timeout*)

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

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

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises `gum.OutOfBounds` – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)
H(self, 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(*self*)
Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(targetlevs) (for all instanciation of target and evs)

Parameters

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

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

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

Returns a Potential for P(targetslevs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*, *variable*)
isTarget(self, 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(*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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

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

setMaxTime(*self*, *timeout*)

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

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

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises `gum.OutOfBounds` – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

softEvidenceNodes (*self*)

Returns the set of nodes with soft evidence

Return type set

targets (*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 (*self*)

Returns True if the verbosity is enabled

Return type bool

4.5.3 Loopy sampling

Loopy Gibbs Sampling

class pyAgrum.**LoopyGibbsSampling** (*bn: pyAgrum.IBayesNet*)
Class used for inferences using a loopy version of Gibbs sampling.

LoopyGibbsSampling(bn) -> LoopyGibbsSampling

Parameters:

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

BN (*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 (*self*, *X*)
H(self, 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 (*self*)
Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)
addTarget(self, 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(*self*)

Returns size of burn in on number of iteration

Return type int

chgEvidence(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*)

Returns True if variables are drawn at random

Return type bool

isTarget(*self*, *variable*)
isTarget(self, 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(*self*)
Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

makeInference_(*self*)

maxIter(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrDrawnVar(*self*)

Returns the number of variable drawn at each iteration

Return type int

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)
posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setBurnIn(*self*, *b*)

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

setDrawnAtRandom(*self*, *_atRandom*)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setNbrDrawnVar(*self*, *_nbr*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*self*, *vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)

H(self, 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(*self*)

Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)

addTarget(self, 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(*self*, *id*, *val*)

chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)

currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

eraseAllEvidence(*self*)

Removes all the evidence entered into the network.

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(targetlevs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targetslevs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*, *variable*)
isTarget(self, 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(*self*)
Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

makeInference_(*self*)

maxIter(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)

posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises `gum.OutOfBounds` – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*self*, *vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)
H(self, 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(*self*)
Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)
addTarget(self, 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(*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)
currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*, *variable*)
isTarget(self, 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(*self*)
Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

makeInference_(*self*)

maxIter(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)

posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises gum.OutOfBounds – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*self*, *vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*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(*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(*self*, *X*)

H(self, 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(*self*)

Add all the nodes as targets.

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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(*self*, *target*)

addTarget(self, 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(*self*, *id*, *val*)

chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*, *id*)

currentPosterior(self, name) -> Potential

Computes and returns the current posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the current posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

Raises *UndefinedElement* (page 257) – If an element of nodes is not in targets

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

epsilon(*self*)

Returns the value of epsilon

Return type double

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

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)
Create a pyAgrum.Potential for P(target|evs) (for all instanciation of target and evs)

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)
hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*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(*self*, *variable*)
isTarget(self, 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(*self*)
Perform the heavy computations needed to compute the targets' posteriors

In a Junction tree propagation scheme, for instance, the heavy computations are those of the messages sent in the JT. This is precisely what makeInference should compute. Later, the computations of the posteriors can be done 'lightly' by multiplying and projecting those messages.

makeInference_(*self*)

maxIter(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrEvidence(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

posterior(*self*, *var*)

posterior(self, nodeName) -> Potential

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

setEpsilon(*self*, *eps*)

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

Raises gum.OutOfBounds – 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(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – 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(*self*, *v*)

Parameters **v** (*bool*) – verbosity

setVirtualLBPSize(*self*, *vlbpsize*)

Parameters **vlbpsize** (*double*) – the size of the virtual LBP

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*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(*self*)

Returns True if the verbosity is enabled

Return type bool

4.6 Learning

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

class pyAgrum.**BNLearner**(*filename*, *inducedTypes=True*)

Parameters:

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

BNLearner(filename,src) -> BNLearner

Parameters:

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

BNLearner(learner) -> BNLearner

Parameters:

- **learner** (*pyAgrum.BNLearner*) – the BNLearner to copy

G2(*self*, *var1*, *var2*, *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(*self*, *arc*)

addForbiddenArc(self, tail, head) addForbiddenArc(self, tail, head)

The arc in parameters won't be added.

Parameters

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

addMandatoryArc(*self*, *arc*)
addMandatoryArc(self, tail, head) addMandatoryArc(self, tail, head)

Allow to add prior structural knowledge.

Parameters

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

Raises gum.InvalidDirectedCycle – If the added arc creates a directed cycle in the DAG

addPossibleEdge(*self*, *edge*)
addPossibleEdge(self, tail, head) addPossibleEdge(self, tail, head)

chi2(*self*, *var1*, *var2*, *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(*self*)

Returns get the current running time in second (double)

Return type double

databaseWeight(*self*)

domainSize(*self*, *var*)
domainSize(self, var) -> int

epsilon(*self*)

Returns the value of epsilon

Return type double

eraseForbiddenArc(*self*, *arc*)
eraseForbiddenArc(self, tail, head) eraseForbiddenArc(self, tail, 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(*self*, *arc*)
eraseMandatoryArc(self, tail, head) eraseMandatoryArc(self, tail, 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(*self*, *edge*)
erasePossibleEdge(self, tail, head) erasePossibleEdge(self, tail, 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(*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(*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(*self*, *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(*self*)
latentVariables(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(*self*)
learn a BayesNet from a file (must have read the db before)

Returns the learned BayesNet

Return type *pyAgrum.BayesNet* (page 48)

learnDAG(*self*)
learn a structure from a file

Returns the learned DAG

Return type *pyAgrum.DAG* (page 7)

learnMixedStructure(*self*)

Warning: learner must be using 3off2 or MIIC algorithm

Returns the learned structure as an EssentialGraph

Return type *pyAgrum.EssentialGraph* (page 66)

learnParameters(*self*, *dag*, *takeIntoAccountScore=True*)
learnParameters(self, take_into_account_score=True) -> BayesNet

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

Parameters

- **dag** (pyAgrum.DAG (page 7)) –
- **bn** (pyAgrum.BayesNet (page 48)) –
- **take_into_account_score** (*bool*) – The dag passed in argument may have been learnt from a structure learning. In this case, if the score used to learn the structure has an implicit apriori (like K2 which has a 1-smoothing apriori), it is important to also take into account this implicit apriori for parameter learning. By default, if a score exists, we will learn parameters by taking into account the apriori specified by methods useAprioriXXX () + the implicit apriori of the score, else we just take into account the apriori specified by useAprioriXXX ()

Returns the learned BayesNet

Return type *pyAgrum.BayesNet* (page 48)

Raises

- gum.MissingVariableInDatabase – If a variable of the BN is not found in the database
- gum.UnknownLabelInDatabase – If a label is found in the database that do not correspond to the variable

logLikelihood(*self*, *vars*, *knowing={}*)
logLikelihood(self, vars) -> double logLikelihood(self, vars, knowing={}) -> double logLikelihood(self, 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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nameFromId(*self*, *id*)

Parameters **id** – a node id

Returns the variable's name

Return type str

names(*self*)

Returns the names of the variables in the database

Return type List[str]

nbCols(*self*)
Return the nimber of columns in the database

Returns the number of columns in the database

Return type int

nbRows(*self*)
Return the number of row in the database

Returns the number of rows in the database

Return type int

nbrIterations(*self*)

Returns the number of iterations

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

pseudoCount(*vars*)
access to pseudo-count (priors taken into account)

Parameters **vars** (*list[str]*) – a list of name of vars to add in the pseudo_count

Returns

Return type a Potential containing this pseudo-counts

rawPseudoCount(*self*, *vars*)
rawPseudoCount(self, vars) -> Vector

recordWeight(*self*, *i*)

setAprioriWeight(*weight*)
Deprecated methods in BNLearner for pyAgrum>0.14.0

setDatabaseWeight(*self*, *new_weight*)
Set the database weight which is given as an equivalent sample size.

Parameters **weight** (*double*) – the database weight

setEpsilon(*self*, *eps*)

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

Raises gum.OutOfBounds – If eps<0

setInitialDAG(*self*, *g*)

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

setMaxIndegree(*self*, *max_indegree*)

setMaxIter(*self*, *max*)

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

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

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

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

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

setPeriodSize(*self*, *p*)

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

Raises gum.OutOfBounds – If p<1

setPossibleSkeleton(*self*, *skeleton*)

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

setSliceOrder(*self*, *l*)
setSliceOrder(self, slice_order) setSliceOrder(self, slices)

Set a partial order on the nodes.

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

setVerbosity(*self*, *v*)

Parameters **v** (*bool*) – verbosity

state(*self*)

use3off2(*self*)
Indicate that we wish to use 3off2.

useAprioriBDeu(*self*, *weight=1*)
useAprioriBDeu(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(*self*, *filename*, *weight=1*)
useAprioriDirichlet(self, filename)

useAprioriSmoothing(*self*, *weight=1*)
useAprioriSmoothing(self)

useEM(*self*, *epsilon*)
Indicates if we use EM for parameter learning.

Parameters **epsilon** (*double*) – if epsilon=0.0 then EM is not used if epsilon>0 then EM is used and stops when the sum of the cumulative squared error on parameters is les than epsilon.

useGreedyHillClimbing(*self*)

useK2(*self*, *l*)
useK2(self, order) useK2(self, order)

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

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

useLocalSearchWithTabuList(*self*, *tabu_size=100*, *nb_decrease=2*)
useLocalSearchWithTabuList(self, tabu_size=100) useLocalSearchWithTabuList(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

useMDLCorrection(*self*)
Indicate that we wish to use the MDL correction for 3off2 or MIIC

useMIIC(*self*)
Indicate that we wish to use MIIC.

useNMLCorrection(*self*)
Indicate that we wish to use the NML correction for 3off2 or MIIC

useNoApriori(*self*)

useNoCorrection(*self*)
Indicate that we wish to use the NoCorr correction for 3off2 or MIIC

useScoreAIC(*self*)

useScoreBD(*self*)

useScoreBDeu(*self*)

useScoreBIC(*self*)

useScoreK2(*self*)

useScoreLog2Likelihood(*self*)

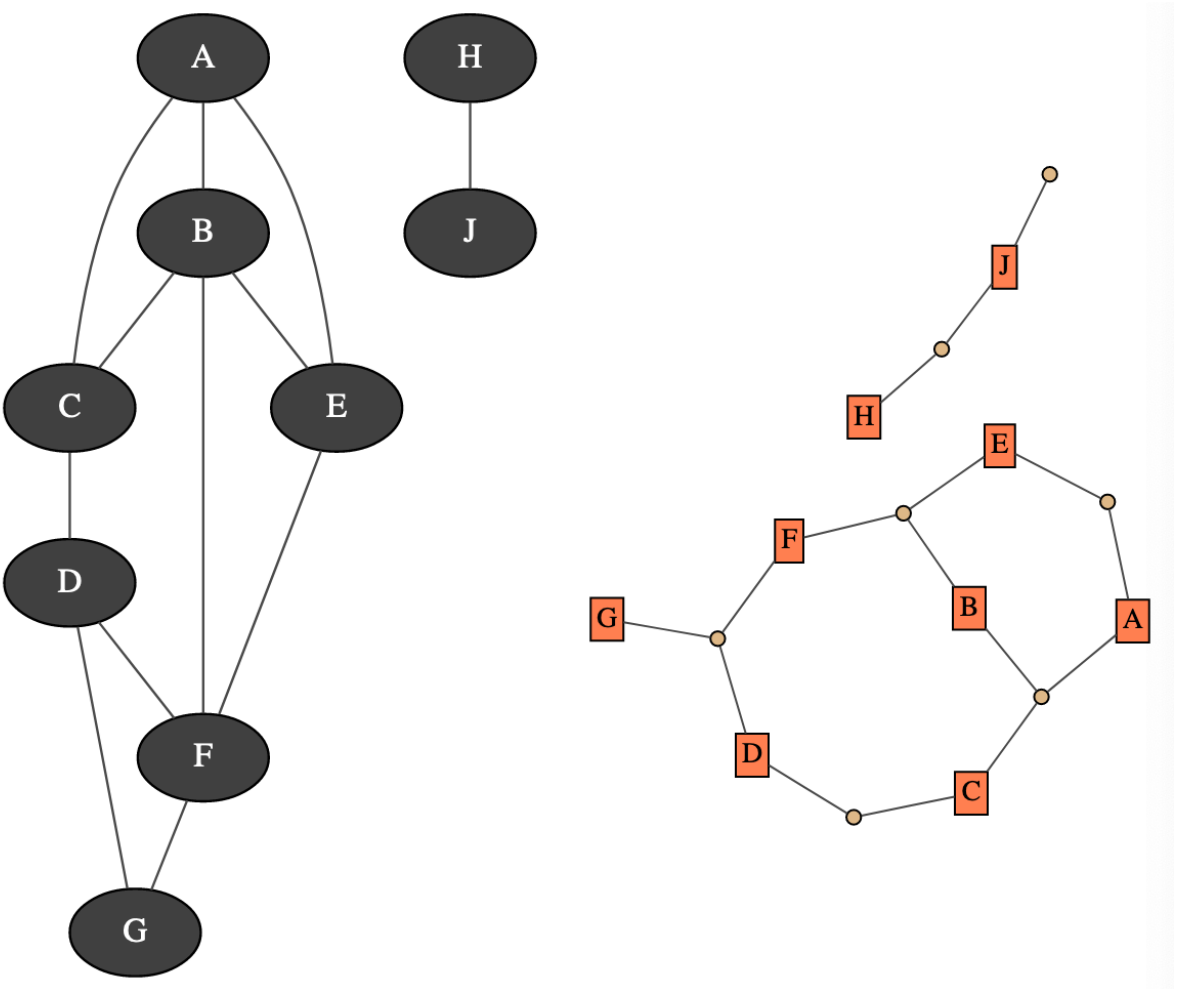
verbosity(*self*)

Returns True if the verbosity is enabled

Return type bool

CHAPTER 5

Markov Network



A Markov network is a undirected probabilistic graphical model. It represents a joint distribution over a set of random variables. In pyAgrum, the variables are (for now) only discrete.

A Markov network uses a undirected graph to represent conditional independence in the joint distribution. These conditional independence allow to factorize the joint distribution, thereby allowing to compactly represent very

large ones.

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

Where the ϕ_i are potentials over the n_c cliques of the undirected graph.

Moreover, inference algorithms can also use this graph to speed up the computations.

Tutorial

- Tutorial on Markov Network (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/current/notebooks/13_Models-MarkovNetwork.ipynb.html)

Reference

5.1 Model

class pyAgrum.**MarkovNet**(**args*)

MarkovNet represents a Markov Network.

MarkovNet(name='') -> MarkovNet

Parameters:

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

MarkovNet(source) -> MarkovNet

Parameters:

- **source** (*pyAgrum.MarkovNet*) – the Markov network to copy

add(*self*, *var*)

add(self, name, nbrmod) -> int add(self, var, id) -> int

Add a variable to the pyAgrum.MarkovNet.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable added
- **name** (*str*) – the variable name
- **nbrmod** (*int*) – the number of modalities for the new variable
- **id** (*int*) – the variable forced id in the pyAgrum.MarkovNet

Returns the id of the new node

Return type int

Raises

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

addFactor(*self*, *varnames*)

addFactor(self, vars) -> Potential addFactor(self, factor) -> Potential addFactor(self, seq) -> Potential

addStructureListener(*whenNodeAdded=None*, *whenNodeDeleted=None*, *whenEdgeAdded=None*, *whenedgeDeleted=None*)

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

Parameters

- **whenNodeAdded** (*lambda expression*) – a function for when a node is added
- **whenNodeDeleted** (*lambda expression*) – a function for when a node is removed
- **whenEdgeAdded** (*lambda expression*) – a function for when an edge is added
- **whenEdgeDeleted** (*lambda expression*) – a function for when an edge is removed

beginTopologyTransformation(*self*)

changeVariableLabel(*self*, *id*, *old_label*, *new_label*)
changeVariableLabel(self, name, old_label, 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(*self*, *id*, *new_name*)
changeVariableName(self, name, new_name)

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

This will change the pyAgrum.DiscreteVariable names in the pyAgrum.MarkovNet.

Parameters

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

Raises

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

clear(*self*)
Clear the whole MarkovNet

completeInstantiation(*self*)

connectedComponents()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

dim(*self*)

edges(*self*)

empty(*self*)

endTopologyTransformation(*self*)
Terminates a sequence of insertions/deletions of arcs by adjusting all CPTs dimensions. End Multiple Change for all CPTs.

Returns

Return type *pyAgrum.MarkovNet* (page 158)

erase(*self*, *varId*)
erase(self, name) erase(self, var)

Remove a variable from the pyAgrum.MarkovNet.

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

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

Parameters

- **id** (*int*) – The variable's id to remove.
- **name** (*str*) – The variable's name to remove.
- **var** (pyAgrum.DiscreteVariable (page 21)) – A reference on the variable to remove.

eraseFactor(*self*, *vars*)
eraseFactor(self, varnames) eraseFactor(self, seq)

exists(*self*, *node*)

existsEdge(*self*, *node1*, *node2*)
existsEdge(self, name1, name2) -> bool

factor(*self*, *varIds*)
factor(self, varnames) -> Potential factor(self, nodeseq) -> Potential

Returns the factor of a set of variables (if existing).

Parameters

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

Returns The factor of the set of nodes.

Return type *pyAgrum.Potential* (page 39)

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

factors(*self*)

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

Create a Markov network with a modified dot-like syntax which specifies:

- the structure `a-b-c;b-d-e;`. The substring `a-b-c` indicates a factor with the scope (a,b,c).
- the type of the variables with different syntax (cf documentation).

Examples

```
>>> import pyAgrum as gum
>>> bn=gum.MarkovNet.fastPrototype('A--B[1,3]-C{yes|No}--D[2,4]--E[1,2.5,3.
↪9]',6)
```

Parameters

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

Returns the resulting Markov network

Return type *pyAgrum.MarkovNet* (page 158)

static fromBN(*bn*)

generateFactor(*self*, *vars*)
Randomly generate factor parameters for a given factor in a given structure.

Parameters

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

generateFactors(*self*)
Randomly generates factors parameters for a given structure.

graph(*self*)

hasSameStructure(*self*, *other*)

idFromName(*self*, *name*)

ids(*self*, *names*)

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

loadUAI(*self*, *name*, *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(*self*)

maxNonOneParam(*self*)

maxParam(*self*)

maxVarDomainSize(*self*)

minNonZeroParam(*self*)

minParam(*self*)

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

names(*self*)

neighbours(*self*, *norid*)

nodeId(*self*, *var*)

nodes(*self*)

nodeset(*self*, *names*)

property(*self*, *name*)

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

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

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

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

size (*self*)

sizeEdges (*self*)

smallestFactorFromNode (*self*, *node*)

toDot (*self*)

toDotAsFactorGraph (*self*)

variable (*self*, *id*)
variable(self, name) -> DiscreteVariable

variableFromName (*self*, *name*)

variableNodeMap (*self*)

5.2 Inference

Inference is the process that consists in computing new probabilistc information from a Markov network and some evidence. aGrUM/pyAgrum mainly focus and the computation of (joint) posterior for some variables of the Markov networks given soft or hard evidence that are the form of likelihoods on some variables. Inference is a hard task (NP-complete). For now, aGrUM/pyAgrum implements only one exact inference for Markov Network.

5.2.1 Shafer Shenoy Inference

class pyAgrum.**ShaferShenoyMNInference** (*MN: pyAgrum.IMarkovNet*, *use_binary_join_tree: bool = True*)
Class used for Shafer-Shenoy inferences for Markov network.

ShaferShenoyInference(bn) -> ShaferShenoyInference

Parameters:

- **mn** (*pyAgrum.MarkovNet*) – a Markov network

H (*self*, *X*)
H(self, nodeName) -> double

Parameters

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

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

Return type double

I (*self*, *X*, *Y*)

Parameters

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

Returns the Mutual Information of X and Y given the observation

Return type double

MN (*self*)

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

Parameters

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

Returns variation of information between X and Y

Return type double

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

addEvidence (*self*, *id*, *val*)
addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, 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 (*self*, *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 (*self*, *target*)
addTarget(self, 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 (*self*, *id*, *val*)
chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, 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(*self*)
Removes all the evidence entered into the network.

eraseAllJointTargets(*self*)
Clear all previously defined joint targets.

eraseAllMarginalTargets(*self*)
Clear all the previously defined marginal targets.

eraseAllTargets(*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(*self*, *id*)
eraseEvidence(self, 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(*self*, *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(*self*, *target*)
eraseTarget(self, 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(*self*, *target*, *evs*)

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

Parameters

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

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

Returns a Potential for P(targets|evs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceJointImpact(self, targets, evs) -> Potential

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

Parameters

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

Returns a Potential for P(target|evs)

Return type *pyAgrum.Potential* (page 39)

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

evidenceProbability(*self*)

Returns the probability of evidence

Return type double

hardEvidenceNodes(*self*)

Returns the set of nodes with hard evidence

Return type set

hasEvidence(*self*, *id*)

hasEvidence(self, 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(*self*, *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(*self*, *id*)
hasSoftEvidence(self, 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(*self*, *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(*self*, *variable*)
isTarget(self, 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(*self*)

Returns the current join tree used

Return type *pyAgrum.CliqueGraph* (page 12)

jointMutualInformation(*self*, *targets*)

jointPosterior(*self*, *targets*)
Compute the joint posterior of a set of nodes.

Parameters **list** – the list of nodes whose posterior joint probability is wanted

Warning: The order of the variables given by the list here or when the jointTarget is declared can not be assumed to be used bu the Potential.

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

Return type *pyAgrum.Potential* (page 39)

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

jointTargets(*self*)

Returns the list of target sets

Return type list

junctionTree(*self*)

Returns the current junction tree

Return type *pyAgrum.CliqueGraph* (page 12)

makeInference(*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(*self*)

Returns the number of evidence entered into the Bayesian network

Return type int

nbrHardEvidence(*self*)

Returns the number of hard evidence entered into the Bayesian network

Return type int

nbrJointTargets(*self*)

Returns the number of joint targets

Return type int

nbrSoftEvidence(*self*)

Returns the number of soft evidence entered into the Bayesian network

Return type int

nbrTargets(*self*)

Returns the number of marginal targets

Return type int

posterior(*self*, *var*)

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

Computes and returns the posterior of a node.

Parameters

- **var** (*int*) – the node Id of the node for which we need a posterior probability
- **nodeName** (*str*) – the node name of the node for which we need a posterior probability

Returns a ref to the posterior probability of the node

Return type *pyAgrum.Potential* (page 39)

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

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

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

Raises

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

setTargets(*targets*)
Remove all the targets and add the ones in parameter.

Parameters **targets** (*set*) – a set of targets

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

setTriangulation(*self*, *new_triangulation*)

softEvidenceNodes(*self*)

Returns the set of nodes with soft evidence

Return type set

targets(*self*)

Returns the list of marginal targets

Return type list

updateEvidence(*evidces*)
Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

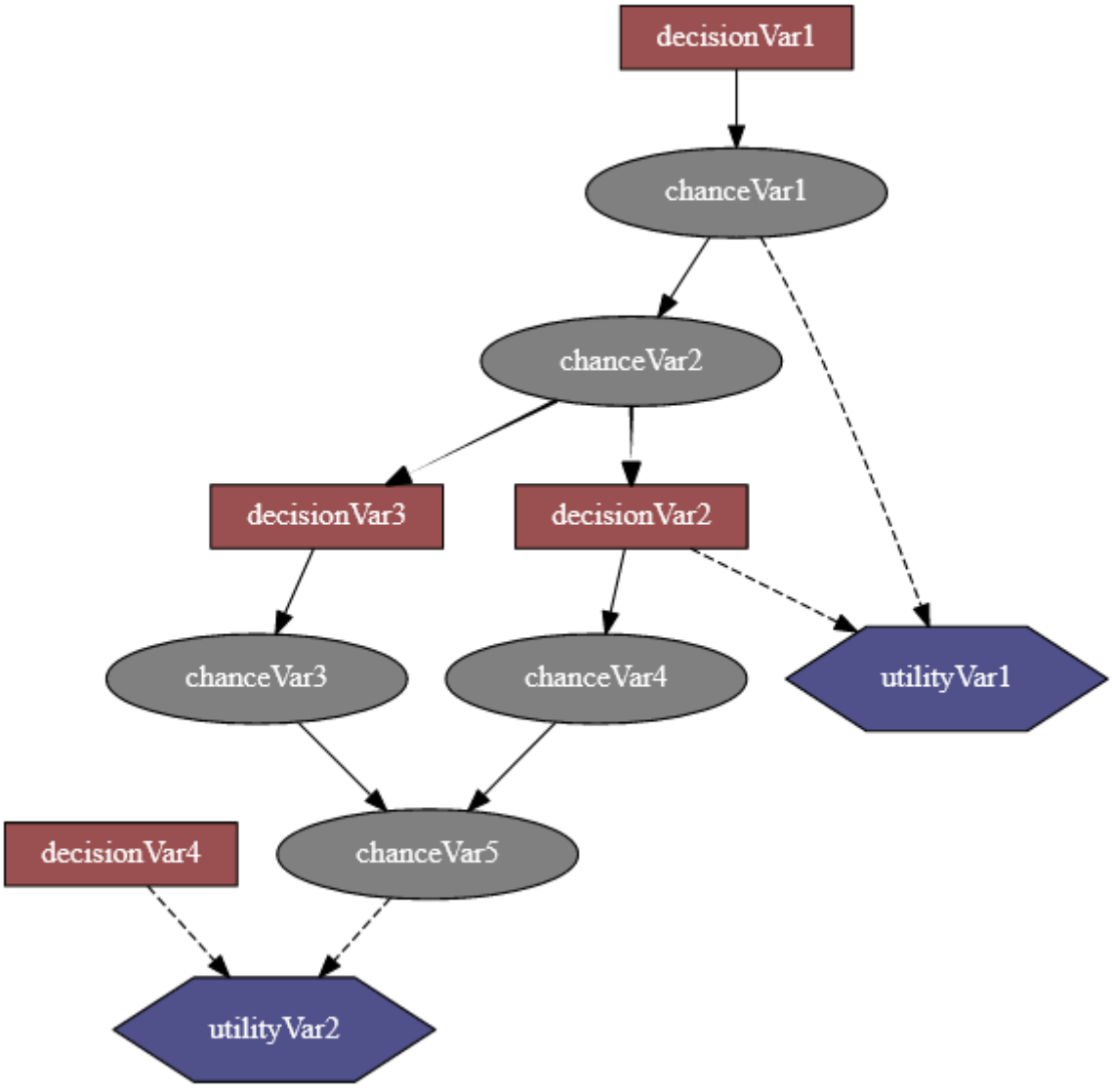
Parameters **evidces** (*dict*) – a dict of evidences

Raises

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

CHAPTER 6

Influence Diagram



An influence diagram is a compact graphical and mathematical representation of a decision situation. It is a generalization of a Bayesian network, in which not only probabilistic inference problems but also decision making problems (following the maximum expected utility criterion) can be modeled and solved. It includes 3 types of nodes : action, decision and utility nodes (from wikipedia (https://en.wikipedia.org/wiki/Influence_diagram)).

PyAgrum's so-called influence diagram represents both influence diagrams and LIMIDs. The way to enforce that such a model represent an influence diagram and not a LIMID belongs to the inference engine.

Tutorial

- Tutorial on Influence Diagram (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/InfluenceDiagram.ipynb.ht

Reference

6.1 Model

class pyAgrum.**InfluenceDiagram**(**args*)

InfluenceDiagram represents an Influence Diagram.

InfluenceDiagram() -> InfluenceDiagram default constructor

InfluenceDiagram(source) -> InfluenceDiagram

Parameters:

- **source** (*pyAgrum.InfluenceDiagram*) – the InfluenceDiagram to copy

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

Add a chance variable, it's associate node and it's CPT.

The id of the new variable is automatically generated.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – The variable added by copy.
- **id** (*int*) – The chosen id. If 0, the NodeGraphPart will choose.

Warning: give an id (not 0) should be reserved for rare and specific situations !!!

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id(<>0) is already used

addArc(*self*, *tail*, *head*)

addArc(self, tail, 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(*self*, *variable*, *id=0*)
addChanceNode(self, variable, aContent, id=0) -> int

Add a chance variable, it's associate node and it's CPT.

The id of the new variable is automatically generated.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable added by copy.
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose.

Warning: give an id (not 0) should be reserved for rare and specific situations !!!

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id(<>0) is already used

addDecisionNode(*self*, *variable*, *id=0*)
Add a decision variable.

The id of the new variable is automatically generated.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable added by copy.
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose.

Warning: give an id (not 0) should be reserved for rare and specific situations !!!

Returns the id of the added variable.

Return type int

Raises gum.DuplicateElement – If id(<>0) is already used

addUtilityNode(*self*, *variable*, *id=0*)
addUtilityNode(self, variable, aContent, id=0) -> int

Add a utility variable, it's associate node and it's UT.

The id of the new variable is automatically generated.

Parameters

- **variable** (pyAgrum.DiscreteVariable (page 21)) – the variable added by copy
- **id** (*int*) – the chosen id. If 0, the NodeGraphPart will choose

Warning: give an id (not 0) should be reserved for rare and specific situations !!!

Returns the id of the added variable.

Return type int

Raises

- `gum.InvalidArgument` – If variable has more than one label
- `gum.DuplicateElement` – If id(<>0) is already used

ancestors (*self*, *norid*)

arcs (*self*)

Returns the list of all the arcs in the Influence Diagram.

Return type list

chanceNodeSize (*self*)

Returns the number of chance nodes.

Return type int

changeVariableName (*self*, *id*, *new_name*)
changeVariableName(self, name, 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 (*self*, *norid*)

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

Returns the set of all the children

Return type Set

clear (*self*)

completeInstantiation (*self*)

connectedComponents ()
connected components from a graph/BN

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

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

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

Return type dict(int,Set[int])

cpt (*self*, *varId*)
cpt(self, name) -> Potential

Returns the CPT of a variable.

Parameters **VarId** (*int*) – A variable's id in the pyAgrum.BayesNet.

Returns The variable's CPT.

Return type *pyAgrum.Potential* (page 39)

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

dag (*self*)

Returns a constant reference to the dag of this BayesNet.

Return type *pyAgrum.DAG* (page 7)

decisionNodeSize(*self*)

Returns the number of decision nodes

Return type int

decisionOrder(*self*)

decisionOrderExists(*self*)

Returns True if a directed path exist with all decision node

Return type bool

descendants(*self*, *norid*)

empty(*self*)

erase(*self*, *id*)
erase(self, name) erase(self, var)

Erase a Variable from the network and remove the variable from all his childs.

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

Parameters

- **id** (*int*) – The id of the variable to erase.
- **var** (pyAgrum.DiscreteVariable (page 21)) – The reference on the variable to remove.

eraseArc(*self*, *arc*)
eraseArc(self, tail, head) eraseArc(self, tail, head)

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

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

Parameters

- **arc** (pyAgrum.Arc (page 3)) – The arc to be removed.
- **tail** (*int*) – the id of the tail node
- **head** (*int*) – the id of the head node

exists(*self*, *node*)

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

existsPathBetween(*self*, *src*, *dest*)
existsPathBetween(self, src, dest) -> bool

Returns true if a path exists between two nodes.

Return type bool

family(*self*, *norid*)

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

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

- the structure 'a->b<-c;b->d;c<-e;'.
- a prefix for the type of node (chance/decision/utiliy nodes):
  - *a* : a chance node named 'a' (by default)
  - *$a* : a utility node named 'a'
  - **a* : a decision node named 'a'

- the type of the variables with different syntax as postfix:
  - by default, a variable is a gum.RangeVariable using the default domain size (second argument)
  - with *'a[10]'*, the variable is a gum.RangeVariable using 10 as domain size (from 0 to 9)
  - with *'a[3,7]'*, the variable is a gum.RangeVariable using a domainSize from 3 to 7
  - with *'a[1,3.14,5,6.2]'*, the variable is a gum.DiscretizedVariable using the given ticks (at least 3 values)
  - with *'a{top|middle|bottom}'*, the variable is a gum.LabelizedVariable using the given labels.
  - with 'a{-1|5|0|3}', the variable is a gum.IntegerVariable using the sorted given values.

Note:

- If the dot-like string contains such a specification more than once for a variable, the first specification will be used.
- the potentials (probabilities, utilities) are randomly generated.
- see also pyAgrum.fastID.

Examples

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

Parameters

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

Returns the resulting Influence Diagram

Return type *pyAgrum.InfluenceDiagram* (page 170)

getDecisionGraph(*self*)

Returns the temporal Graph.

Return type *pyAgrum.DAG* (page 7)

hasSameStructure(*self*, *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(*self*, *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(*self*, *names*)

isChanceNode(*self*, *varId*)
isChanceNode(self, name) -> bool

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

Returns true if node is a chance node

Return type bool

isDecisionNode(*self*, *varId*)
isDecisionNode(self, name) -> bool

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

Returns true if node is a decision node

Return type bool

isIndependent(*self*, *X*, *Y*, *Z*)
isIndependent(self, X, Y, Z) -> bool isIndependent(self, Xname, Yname, Znames) -> bool isIndependent(self, Xnames, Ynames, Znames) -> bool

isUtilityNode(*self*, *varId*)
isUtilityNode(self, name) -> bool

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

Returns true if node is an utility node

Return type bool

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

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

Raises

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

log10DomainSize(*self*)

moralGraph(*self*, *clear=True*)
Returns the moral graph of the BayesNet, formed by adding edges between all pairs of nodes that have a common child, and then making all edges in the graph undirected.

Returns The moral graph

Return type *pyAgrum.UndiGraph* (page 9)

moralizedAncestralGraph(*self*, *nodes*)

names(*self*)

Returns The names of the InfluenceDiagram variables

Return type list

nodeId(*self*, *var*)

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

Returns the id of the variable

Return type int

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

nodes(*self*)

Returns the set of ids

Return type set

nodeset(*self*, *names*)

parents(*self*, *norid*)

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

Returns the set of the parents ids.

Return type set

property(*self*, *name*)

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

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

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

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

size(*self*)

Returns the number of nodes in the graph

Return type int

sizeArcs(*self*)

Returns the number of arcs in the graph

Return type int

toDot(*self*)

Returns a friendly display of the graph in DOT format

Return type str

topologicalOrder(*self*, *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(*self*, *varId*)
utility(self, name) -> Potential

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

Returns the utility table of the node

Return type *pyAgrum.Potential* (page 39)

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

utilityNodeSize(*self*)

Returns the number of utility nodes

Return type int

variable(*self*, *id*)
variable(self, name) -> DiscreteVariable

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

Returns a constant reference over a variabe given it's node id

Return type *pyAgrum.DiscreteVariable* (page 21)

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

variableFromName(*self*, *name*)

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

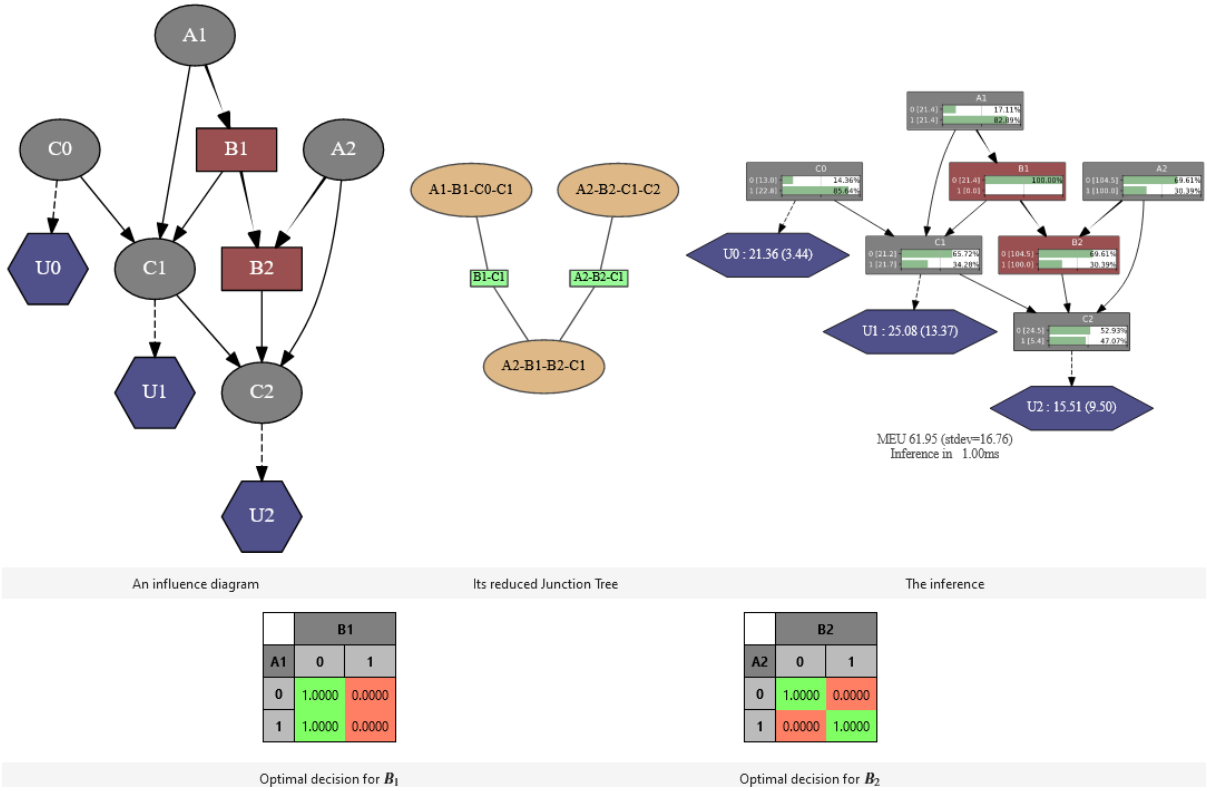
Returns the variable

Return type *pyAgrum.DiscreteVariable* (page 21)

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

variableNodeMap(*self*)

6.2 Inference



	B1	
A1	0	1
0	1.0000	0.0000
1	1.0000	0.0000

Optimal decision for B_1

	B2	
A2	0	1
0	1.0000	0.0000
1	0.0000	1.0000

Optimal decision for B_2

class pyAgrum.**ShaferShenoyLIMIDInference**(*infDiag: pyAgrum.InfluenceDiagram*)

This inference considers the provided model as a LIMID rather than an influence diagram. It is an optimized implementation of the LIMID resolution algorithm. However an inference on a classical influence diagram can be performed by adding a assumption of the existence of the sequence of decision nodes to be solved, which also implies that the decision choices can have an impact on the rest of the sequence (Non Forgetting Assumption, cf. pyAgrum.ShaferShenoyLIMIDInference.addNoForgettingAssumption).

MEU(*self*)

MEU(self) -> PyObject *

Returns maximum expected utility obtained from inference.

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

addEvidence(*self*, *id*, *val*)

addEvidence(self, nodeName, val) addEvidence(self, id, val) addEvidence(self, nodeName, val) addEvidence(self, id, vals) addEvidence(self, nodeName, vals)

addNoForgettingAssumption(*self*, *ids*)

addNoForgettingAssumption(self, names)

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

chgEvidence(self, nodeName, val) chgEvidence(self, id, val) chgEvidence(self, nodeName, val) chgEvidence(self, id, vals) chgEvidence(self, nodeName, vals)

clear(*self*)

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

eraseEvidence(*self*, *id*)
eraseEvidence(self, nodeName)

Parameters **evidence** (pyAgrum.Potential (page 39)) – the evidence to remove

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

hardEvidenceNodes(*self*)

hasEvidence(*self*, *id*)
hasEvidence(self, nodeName) -> bool

hasHardEvidence(*self*, *nodeName*)

hasNoForgettingAssumption(*self*)

hasSoftEvidence(*self*, *id*)
hasSoftEvidence(self, nodeName) -> bool

influenceDiagram(*self*)
Returns a constant reference over the InfluenceDiagram on which this class work.

Returns the InfluenceDiagram on which this class work

Return type *pyAgrum.InfluenceDiagram* (page 170)

isSolvable(*self*)

junctionTree(*self*)

makeInference(*self*)
Makes the inference.

meanVar(*self*, *node*)
meanVar(self, name) -> pair< double,double > meanVar(self, node) -> PyObject *

nbrEvidence(*self*)

nbrHardEvidence(*self*)

nbrSoftEvidence(*self*)

optimalDecision(*self*, *decisionId*)
optimalDecision(self, decisionName) -> Potential

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

Parameters **decisionId** (*int*, *str*) – the id or name of the decision variable

Raises

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

posterior(*self*, *node*)
posterior(self, name) -> Potential posterior(self, var) -> Potential posterior(self, nodeName) -> Potential

posteriorUtility(*self*, *node*)
posteriorUtility(self, name) -> Potential

reducedGraph(*self*)

reducedLIMID(*self*)

reversePartialOrder(*self*)

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

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

Raises

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

softEvidenceNodes(*self*)

updateEvidence(*evidces*)
Apply chgEvidence(key,value) for every pairs in evidces (or addEvidence).

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

Raises

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

CHAPTER 7

Probabilistic Relational Models

For now, pyAgrum only allows to explore Probabilistic Relational Models written with o3prm syntax (see O3PRM website (https:://o3prm.gitlab.io/)).

class pyAgrum.**PRMexplorer**

PRMexplorer helps navigate through probabilistic relational models.

PRMexplorer() -> PRMexplorer default constructor

aggType

min/max/count/exists/forall/or/and/amplitude/median

classAggregates(*self*, *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(*self*, *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(*self*, *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(*self*, *class_name*)

Parameters **class_name** (*str*) – a class name

Returns the list of interfaces implemented by the class

Return type list

classParameters(*self*, *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(*self*, *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(*self*, *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(*self*)

Returns the list of classes

Return type list

cpf(*self*, *class_name*, *attribute*)

Parameters

- **class_name** (*str*) – a class name
- **attribute** (*str*) – an attribute

Returns the potential of the attribute

Return type *pyAgrum.Potential* (page 39)

Raises

- gum.OperationNotAllowed – If the class element doesn't have any pyAgrum.Potential (like a pyAgrum.PRMReferenceSlot).
- gum.IndexError – If the class is not in the PRM
- gum.IndexError – If the attribute in parameters does not exist

getDirectSubClass(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*)

Returns the list of all the systems and their components

Return type list

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

Parameters

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

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

Return type list

Raises gum.IndexError – If the type is not in the PRM

interReferences(*self*, *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(*self*)

Returns the list of interfaces in the PRM

Return type list

isAttribute(*self*, *class_name*, *att_name*)

Parameters

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

Returns True if att_name is an attribute of class_name

Return type bool

Raises

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

isClass(*self*, *name*)

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

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

Return type bool

isInterface(*self*, *name*)

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

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

Return type bool

isType(*self*, *name*)

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

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

Return type bool

load(*self*, *filename*, *classpath=""*, *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(*self*)

Returns the list of the custom types in the PRM

Return type list

CHAPTER 8

Credal Network

Credal networks are probabilistic graphical models based on imprecise probability. Credal networks can be regarded as an extension of Bayesian networks, where credal sets replace probability mass functions in the specification of the local models for the network variables given their parents. As a Bayesian network defines a joint probability mass function over its variables, a credal network defines a joint credal set (from Wikipedia (https://en.wikipedia.org/wiki/Credal_network)).

Tutorial

- Tutorial on Credal Networks (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/credalNetworks.ipynb.html)

Reference

8.1 Model

class pyAgrum.**CredalNet**(**args*)

Constructor used to create a CredalNet (step by step or with two BayesNet)

CredalNet() -> CredalNet default constructor

CredalNet(src_min_num,src_max_den) -> CredalNet

Parameters

- **src_min_num** (*str or* pyAgrum.BayesNet (page 48)) – The path to a BayesNet or the BN itself which contains lower probabilities.
- **src_max_den** (*str or* pyAgrum.BayesNet (page 48)) – The (optional) path to a BayesNet or the BN itself which contains upper probabilities.

addArc(*self*, *tail*, *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(*self*, *name*, *card*)

Parameters

- **name** (*str*) – the name of the new variable
- **card** (*int*) – the domainSize of the new variable

Returns the id of the new node

Return type int

approximatedBinarization(*self*)

Approximate binarization.

Each bit has a lower and upper probability which is the lowest - resp. highest - over all vertices of the credal set. Enlarge the orignal credal sets and may induce huge imprecision.

Warning: Enlarge the orignal credal sets and therefor induce huge imprecision by propagation. Not recommended, use MCSampling or something else instead

bnToCredal(*self*, *beta*, *oneNet*, *keepZeroes=False*)

Perturbates the BayesNet provided as input for this CredalNet by generating intervals instead of point probabilities and then computes each vertex of each credal set.

Parameters

- **beta** (*double*) – The beta used to perturbate the network
- **oneNet** (*bool*) – used as a flag. Set to True if one BayesNet if provided with counts, to False if two BayesNet are provided; one with probabilities (the lower net) and one with denominators over the first modalities (the upper net)
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept

computeBinaryCPTMinMax(*self*)

credalNet_currentCpt(*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(*self*)

Warning: Experimental function - Return type to be wrapped

Returns a constant reference to the (up-to-date) CredalNet CPTs.

Return type tbw

currentNodeType(*self*, *id*)

Parameters **id** (*int*) – The constant reference to the choosen NodeId

Returns the type of the choosen node in the (up-to-date) CredalNet __current_bn if any, __src_bn otherwise.

Return type *pyAgrum.CredalNet* (page 187)

current_bn(*self*)

Returns Returs a constant reference to the actual BayesNet (used as a DAG, it's CPTs does not matter).

Return type *pyAgrum.BayesNet* (page 48)

domainSize(*self*, *id*)

Parameters **id** (*int*) – The id of the node

Returns The cardinality of the node

Return type int

epsilonMax(*self*)

Returns a constant reference to the highest perturbation of the BayesNet provided as input for this CredalNet.

Return type double

epsilonMean(*self*)

Returns a constant reference to the average perturbation of the BayesNet provided as input for this CredalNet.

Return type double

epsilonMin(*self*)

Returns a constant reference to the lowest perturbation of the BayesNet provided as input for this CredalNet.

Return type double

fillConstraint(*self*, *id*, *entry*, *lower*, *upper*)
fillConstraint(self, id, ins, lower, upper)

Set the interval constraints of a credal set of a given node (from an instantiation index)

Parameters

- **id** (*int*) – The id of the node
- **entry** (*int*) – The index of the instantiation excluding the given node (only the parents are used to compute the index of the credal set)
- **ins** (pyAgrum.Instantiation (page 34)) – The Instantiation
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

Warning: You need to call intervalToCredal when done filling all constraints.

Warning: DOES change the BayesNet (s) associated to this credal net !

fillConstraints(*self*, *id*, *lower*, *upper*)
Set the interval constraints of the credal sets of a given node (all instantiations)

Parameters

- **id** (*int*) – The id of the node
- **lower** (*list*) – The lower value for each probability in correct order
- **upper** (*list*) – The upper value for each probability in correct order

Warning: You need to call intervalToCredal when done filling all constraints.

Warning: DOES change the BayesNet (s) associated to this credal net !

get_binaryCPT_max(*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_binaryCPT_min(*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

hasComputedBinaryCPTMinMax(*self*)

idmLearning(*self*, *s=0*, *keepZeroes=False*)
Learns parameters from a BayesNet storing counts of events.

Use this method when using a single BayesNet storing counts of events. IDM model if s > 0, standard point probability if s = 0 (default value if none precised).

Parameters

- **s** (*int*) – the IDM parameter.
- **keepZeroes** (*bool*) – used as a flag as whether or not - respectively True or False - we keep zeroes as zeroes. Default is False, i.e. zeroes are not kept.

instantiation(*self*, *id*)
Get an Instantiation from a node id, usefull to fill the constraints of the network.

bnet accessors / shortcuts.

Parameters **id** (*int*) – the id of the node we want an instantiation from

Returns the instantiation

Return type *pyAgrum.Instantiation* (page 34)

intervalToCredal(*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(*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(*self*)

Returns True if this CredalNet is separately and interval specified, False otherwise.

Return type bool

lagrangeNormalization(*self*)
Normalize counts of a BayesNet storing counts of each events such that no probability is 0.

Use this method when using a single BayesNet storing counts of events. Lagrange normalization. This call is irreversible and modify counts stored by __src_bn.

Doest not performs computations of the parameters but keeps normalized counts of events only. Call idmLearning to compute the probabilities (with any parameter value).

nodeType(*self*, *id*)

Parameters **id** (*int*) – the constant reference to the choosen NodeId

Returns the type of the choosen node in the (up-to-date) CredalNet in __src_bn.

Return type *pyAgrum.CredalNet* (page 187)

saveBNsMinMax(*self*, *min_path*, *max_path*)
If this CredalNet was built over a perturbed BayesNet, one can save the intervals as two BayesNet.

to call after bnToCredal(GUM_SCALAR beta) save a BN with lower probabilities and a BN with upper ones

Parameters

- **min_path** (*str*) – the path to save the BayesNet which contains the lower probabilities of each node X.
- **max_path** (*str*) – the path to save the BayesNet which contains the upper probabilities of each node X.

setCPT(*self*, *id*, *entry*, *cpt*)
setCPT(self, id, ins, cpt)

> **Warning:** (experimental function) - Parameters to be wrapped

Set the vertices of one credal set of a given node (any instantiation index)

Parameters

- **id** (*int*) – the Id of the node
- **entry** (*int*) – the index of the instantiation (from 0 to K - 1) excluding the given node (only the parents are used to compute the index of the credal set)
- **ins** (pyAgrum.Instantiation (page 34)) – the Instantiation (only the parents matter to find the credal set index)
- **cpt** (*tbw*) – the vertices of every credal set (for each instantiation of the parents)

> **Warning:** DOES not change the BayesNet(s) associated to this credal net !

setCPTs (*self*, *id*, *cpt*)

> **Warning:** (experimental function) - Parameters to be wrapped

Set the vertices of the credal sets (all of the conditionals) of a given node

Parameters

- **id** (*int*) – the NodeId of the node
- **cpt** (*tbw*) – the vertices of every credal set (for each instantiation of the parents)

> **Warning:** DOES not change the BayesNet (s) associated to this credal net !

src_bn (*self*)

Returns Returns a constant reference to the original BayesNet (used as a DAG, it's CPTs does not matter).

Return type *pyAgrum.BayesNet* (page 48)

8.2 Inference

class pyAgrum.**CNMonteCarloSampling** (*credalNet: pyAgrum.CredalNet*)

Class used for inferences in credal networks with Monte Carlo sampling algorithm.

CNMonteCarloSampling(cn) -> CNMonteCarloSampling

Parameters:

- **cn** (*pyAgrum.CredalNet*) – a credal network

CN (*self*)

currentTime (*self*)

Returns get the current running time in second (double)

Return type double

dynamicExpMax (*self*, *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 (*self*, *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 (*self*)

Returns the value of epsilon

Return type double

history(*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(*self*, *path*)
Insert evidence from file.

Parameters **path** (*str*) – the path to the evidence file.

insertModalsFile(*self*, *path*)
Insert variables modalities from file to compute expectations.

Parameters **path** (*str*) – The path to the modalities file.

makeInference(*self*)
Starts the inference.

marginalMax(*self*, *id*)
marginalMax(self, name) -> Potential

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(*self*, *id*)
marginalMin(self, name) -> Potential

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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrIterations(*self*)

Returns the number of iterations

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises gum.OutOfBounds – If p<1

setEpsilon(*self*, *eps*)

Parameters **eps** (*double*) – the epsilon we want to use

Raises gum.OutOfBounds – If eps<0

setMaxIter(*self*, *max*)

Parameters **max** (*int*) – the maximum number of iteration

Raises gum.OutOfBounds – If max <= 1

setMaxTime(*self*, *timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*self*, *p*)

Parameters **p** (*int*) – number of samples between 2 stopping

Raises gum.OutOfBounds – If p<1

setRepetitiveInd(*self*, *flag*)

Parameters **flag** (*bool*) – True if repetitive independence is to be used, false otherwise. Only usefull with dynamic networks.

setVerbosity(*self*, *v*)

Parameters **v** (*bool*) – verbosity

verbosity(*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

CN(*self*)

currentTime(*self*)

Returns get the current running time in second (double)

Return type double

dynamicExpMax (*self*, *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 (*self*, *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 (*self*)

Returns the value of epsilon

Return type double

eraseAllEvidence (*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 (*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 (*self*, *inft*)
inferenceType(self) -> pyAgrum.credal::CNLoopyPropagation ::InferenceType

Returns the inference type

Return type int

insertEvidenceFile (*self*, *path*)
Insert evidence from file.

Parameters **path** (*str*) – the path to the evidence file.

insertModalsFile (*self*, *path*)
Insert variables modalities from file to compute expectations.

Parameters **path** (*str*) – The path to the modalities file.

makeInference (*self*)
Starts the inference.

marginalMax (*self*, *id*)
marginalMax(self, name) -> Potential

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(*self*, *id*)
marginalMin(self, name) -> Potential

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(*self*)

Returns the criterion on number of iterations

Return type int

maxTime(*self*)

Returns the timeout(in seconds)

Return type double

messageApproximationScheme(*self*)

Returns the approximation scheme message

Return type str

minEpsilonRate(*self*)

Returns the value of the minimal epsilon rate

Return type double

nbrIterations(*self*)

Returns the number of iterations

Return type int

periodSize(*self*)

Returns the number of samples between 2 stopping

Return type int

Raises `gum.OutOfBounds` – If p<1

saveInference(*self*, *path*)
Saves marginals.

Parameters **path** (*str*) – The path to the file to save marginals.

setEpsilon(*self*, *eps*)

Parameters **eps** (*double*) – the epsilon we want to use

Raises `gum.OutOfBounds` – If eps<0

setMaxIter(*self*, *max*)

Parameters **max** (*int*) – the maximum number of iteration

Raises `gum.OutOfBounds` – If max <= 1

setMaxTime(*self*, *timeout*)

Parameters **tiemout** (*double*) – stopping criterion on timeout (in seconds)

Raises gum.OutOfBounds – If timeout<=0.0

setMinEpsilonRate(*self*, *rate*)

Parameters **rate** (*double*) – the minimal epsilon rate

setPeriodSize(*self*, *p*)

Parameters **p** (*int*) – number of samples between 2 stopping

Raises gum.OutOfBounds – If p<1

setRepetitiveInd(*self*, *flag*)

Parameters **flag** (*bool*) – True if repetitive independence is to be used, false otherwise. Only usefull with dynamic networks.

setVerbosity(*self*, *v*)

Parameters **v** (*bool*) – verbosity

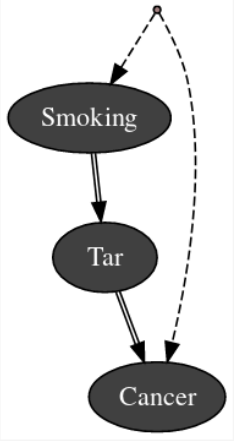
verbosity(*self*)

Returns True if the verbosity is enabled

Return type bool

CHAPTER 9

pyAgrum.causal documentation



$$P(Cancer \mid\hookrightarrow Smoking) = \sum_{Tar} P\left(Tar \mid Smoking\right) \cdot \left(\sum_{Smoking'} P\left(Cancer \mid Smoking', Tar\right) \cdot P\left(Smoking'\right) \right)$$

Causality in pyAgrum mainly consists in the ability to build a causal model, i.e. a (observational) Bayesian network and a set of latent variables and their relation with observation variables and in the abilidy to compute using do-calculus the causal impact in such a model.

Causality is a set of pure python3 scripts based on pyAgrum's tools.

Tutorial

- Notebooks on causality in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Tobacco.ipynb.html)
- Some implemented examples (https://webia.lip6.fr/~phw/aGrUM/BookOfWhy/) from the book of Why (http://bayes.cs.ucla.edu/WHY/) from Judea Pearl and Dana Mackenzie.

Reference

9.1 Causal Model

class pyAgrum.causal.**CausalModel**(*bn: pyAgrum.BayesNet, latentVarsDescriptor: List[Tuple[str, Tuple[str, str]]] = None, keepArcs: bool = False*)

From an observational BNs and the description of latent variables, this class represent a complet causal

model obtained by adding the latent variables specified in `latentVarsDescriptor` to the Bayesian network `bn`.

Parameters

- **bn** – a observational Bayesian network
- **latentVarsDescriptor** – list of couples (<latent variable name>, <list of affected variables' ids>).
- **keepArcs** – By default, the arcs between variables affected by a common latent variable will be removed but this can be avoided by setting `keepArcs` to `True`

causalBN() → pyAgrum.BayesNet

Returns the causal Bayesian network

Warning do not infer any computations in this model. It is strictly a structural model

children(*x: Union[int, str]*) → Set[int]

Parameters **x** – the node

Returns

idFromName(*name: str*) → int

Parameters **name** – the name of the variable

Returns the id of the variable

latentVariablesIds() → Set[int]

Returns the set of ids of latent variables in the causal model

names() → Dict[int, str]

Returns the map NodeId,Name

observationalBN() → pyAgrum.BayesNet

Returns the observational Bayesian network

parents(*x: Union[int, str]*) → Set[int]

From a NodeId, returns its parent (as a set of NodeId)

Parameters **x** – the node

Returns

9.2 Causal Formula

CausalFormula is the class that represents a causal query in a causal model. Mainly it consits in

- a reference to the CausalModel
- Three sets of variables name that represent the 3 sets of variable in the query P(set1 | doing(set2),knowing(set3)).
- the AST for compute the query.

class pyAgrum.causal.**CausalFormula**(*cm: CausalModel, root: pyAgrum.causal._doAST.ASTtree, on: Union[str, Set[str]], doing: Union[str, Set[str]], knowing: Optional[Set[str]] = None*)

Represents a causal query in a causal model. The query is encoded as an CausalFormula that can be evaluated in the causal model : $P(on|knowing,overhook (doing))$

Parameters

- **cm** – the causal model

- **root** – the syntax tree as the root ASTtree
- **on** – the variable or the set of variables of interest
- **doing** – the intervention variables
- **knowing** – the observation variables

cm
return: the causal model

copy() → pyAgrum.causal._CausalFormula.CausalFormula
Copy theAST. Note that the causal model is just referenced. The tree is copied.

Returns the new CausalFormula

eval() → pyAgrum.Potential
Compute the Potential from the CausalFormula over vars using cond as value for others variables

Parameters **bn** – the BN where to infer

Returns

latexQuery(*values: Optional[Dict[str, str]] = None*) → str
Returns a string representing the query compiled by this Formula. If values, the query is annotated with the values in the dictionary.

Parameters **values** – the values to add in the query representation

Returns the string representing the causal query for this CausalFormula

root
return: ASTtree root of the CausalFormula tree

toLatex() → str

Returns a LaTeX representation of the CausalFormula

9.3 Causal Inference

Obtaining and evaluating a CausalFormula is done using one these functions :

pyAgrum.causal.**causalImpact**(*cm: pyAgrum.causal._CausalModel.CausalModel, on: Union[str, Set[str]], doing: Union[str, Set[str]], knowing: Optional[Set[str]] = None, values: Optional[Dict[str, int]] = None*) → Tuple[pyAgrum.causal._CausalFormula.CausalFormula, pyAgrum.Potential, str]
Determines the causal impact of interventions.

Determines the causal impact of the interventions specified in `doing` on the single or list of variables `on` knowing the states of the variables in `knowing` (optional). These last parameters is dictionary <variable name>:<value>. The causal impact is determined in the causal DAG `cm`. This function returns a triplet with a latex format formula used to compute the causal impact, a potential representing the probability distribution of `on` given the interventions and observations as parameters, and an explanation of the method allowing the identification. If there is no impact, the joint probability of `on` is simply returned. If the impact is not identifiable the formula and the adjustment will be `None` but an explanation is still given.

Parameters

- **cm** – causal model
- **on** – variable name or variable names set
- **doing** – variable name or variable names set
- **knowing** – variable names set
- **values** – Dictionary

Returns the CausalFormula, the computation, the explanation

pyAgrum.causal.**doCalculusWithObservation**(*cm: pyAgrum.causal._CausalModel.CausalModel, on: str, doing: Set[str], knowing: Optional[Set[str]] = None*) → pyAgrum.causal._CausalFormula.CausalFormula

Compute the CausalFormula for an impact analysis given the causal model, the observed variables and the variable on which there will be intervention.

Parameters

- **on** – the variables of interest
- **cm** – the causal model
- **doing** – the interventions
- **knowing** – the observations

Returns the CausalFormula for computing this causal impact

pyAgrum.causal.**identifyingIntervention**(*cm: pyAgrum.causal._CausalModel.CausalModel, Y: Set[str], X: Set[str], P: pyAgrum.causal._doAST.ASTtree = None*) → pyAgrum.causal._doAST.ASTtree

Following Shpitser, Ilya and Judea Pearl. 'Identification of Conditional Interventional Distributions.' UAI2006 and 'Complete Identification Methods for the Causal Hierarchy' JMLR 2008

Parameters

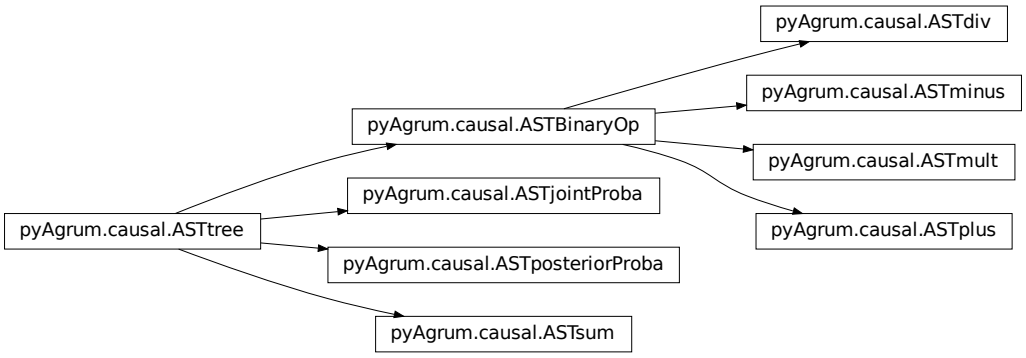
- **cm** – the causal model
- **Y** – The variables of interest (named following the paper)
- **X** – The variable of intervention (named following the paper)
- **P** – The ASTtree representing the calculus in construction

Returns the ASTtree representing the calculus

9.4 Abstract Syntax Tree for Do-Calculus

The pyCausal package compute every causal query into an Abstract Syntax Tree (CausalFormula) that represents the exact computations to be done in order to answer to the probabilistic causal query.

The different types of node in an CausalFormula are presented below and are organized as a hierarchy of classes from *pyAgrum.causal.ASTtree* (page 203).



9.4.1 Internal node structure

class pyAgrum.causal.**ASTtree**(*type: str*, *verbose=False*)
Represents a generic node for the CausalFormula. The type of the node will be registered in a string.

Parameters **type** – the type of the node (will be specified in concrete children classes.

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new causal tree

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTBinaryOp**(*type: str*, *op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)
Represents a generic binary node for the CausalFormula. The op1 and op2 are the two operands of the class.

Parameters

- **type** – the type of the node (will be specified in concrete children classes
- **op1** – left operand
- **op2** – right operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new causal tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

9.4.2 Basic Binary Operations

class pyAgrum.causal.**ASTplus**(*op1: pyAgrum.causal._doAST.ASTtree*, *op2: pyAgrum.causal._doAST.ASTtree*)
Represents the sum of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTminus**(*op1: pyAgrum.causal._doAST.ASTtree, op2: pyAgrum.causal._doAST.ASTtree*)
Represents the substraction of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTdiv**(*op1: pyAgrum.causal._doAST.ASTtree, op2: pyAgrum.causal._doAST.ASTtree*)
Represents the division of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTmult**(*op1: pyAgrum.causal._doAST.ASTtree, op2: pyAgrum.causal._doAST.ASTtree*)
Represents the multiplication of 2 causal.ASTtree

Parameters

- **op1** – first operand
- **op2** – second operand

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

op1
return: the left operand

op2
return: the right operand

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

9.4.3 Complex operations

class pyAgrum.causal.**ASTsum**(*var: List[str], term: pyAgrum.causal._doAST.ASTtree*)
Represents a sum over a variable of a `causal.ASTtree`.

Parameters

- **var** – name of the variable
- **term** – the tree to be evaluated

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

eval(*contextual_bn: pyAgrum.BayesNet*) → pyAgrum.Potential
Evaluation of the sum

Parameters **contextual_bn** – BN where to infer

Returns the value of the sum

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

class pyAgrum.causal.**ASTjointProba**(*varNames: Set[str]*)
Represent a joint probability in the base observational part of the `causal.CausalModel`

Parameters **varNames** – a set of variable names

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

varNames
return: the set of names of var

class pyAgrum.causal.**ASTposteriorProba**(*bn: pyAgrum.BayesNet, vars: Set[str], knw: Set[str]*)
Represent a conditional probability $P_{bn}(vars|knw)$ that can be computed by an inference in a BN.

Parameters

- **bn** – the pyAgrum:pyAgrum.BayesNet
- **vars** – a set of variable names (in the BN)
- **knw** – a set of variable names (in the BN)

bn
return: bn in $P_{bn}(vars|knw)$

copy() → pyAgrum.causal._doAST.ASTtree
Copy an CausalFormula tree

Returns the new CausalFormula tree

knw
return: knw in $P_{bn}(vars|knw)$

toLatex(*nameOccur: Optional[Dict[str, int]] = None*) → str
Create a LaTeX representation of a ASTtree

Returns the LaTeX string

type
return: the type of the node

vars
return: vars in $P_{bn}(vars|knw)$

9.5 Exceptions

class pyAgrum.causal.**HedgeException**(*msg: str, observables: Set[str], gs*)
Represents an hedge exception for a causal query

Parameters

- **msg** – str
- **observables** – NameSet
- **gs** – ???

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pyAgrum.causal.**UnidentifiableException**(*msg*)
Represents an unidentifiability for a causal query

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

9.6 Notebook's tools for causality

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`

CHAPTER 10

pyAgrum.skbn documentation

Probabilistic classification in pyAgrum aims to propose a scikit-learn-like (binary and multi-class) classifier class that can be used in the same codes as scikit-learn classifiers. Moreover, even if the classifier wraps a full Bayesian network, skbn optimally encodes the classifier using the smallest set of needed features following the d-separation criterion (Markov Blanket).

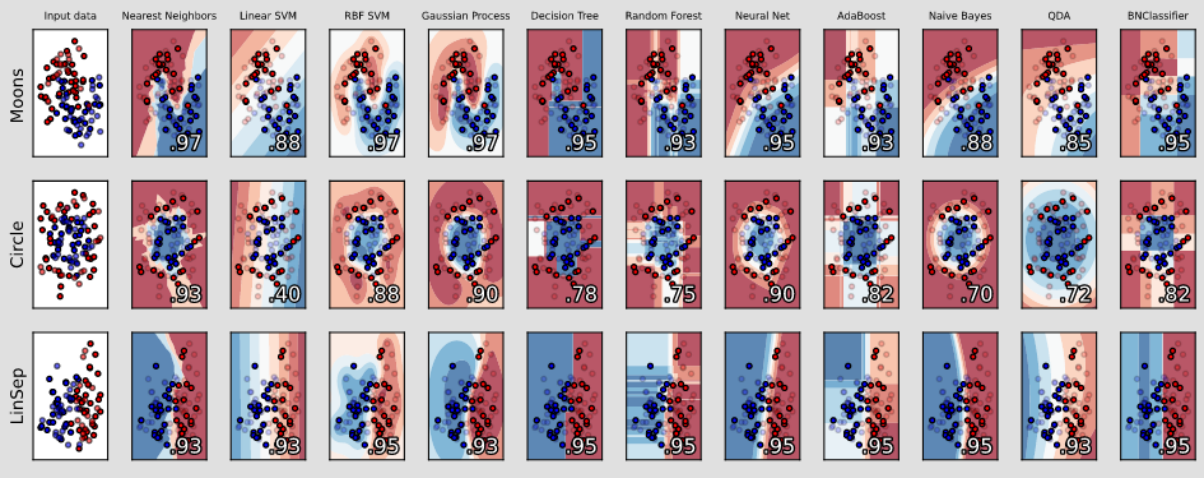


Fig. 1: An example from scikit-learn (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html) where a last column with a BNClassifier has been added flawlessly (see this notebook (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html)).

The module proposes to wrap the pyAgrum's learning algorithms and some others (naive Bayes, TAN, Chow-Liu tree) in the fit method of a classifier. In order to be used with continuous variable, the module proposes also some different discretization methods.

skbn is a set of pure python3 scripts based on pyAgrum's tools.

Tutorials

- Notebooks on scikit-learn-like classifiers in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Learning.ipynb.html), the integration in scikit-learn codes (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CompareClassifiersWithSklearn.ipynb.html) and, as an example, cross-validation with scikit-learn (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/CrossValidation.ipynb.html)

- An example from Kaggle (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/KaggleTitanic.ipynb.html),
- Notebook on Discretizers in pyAgrum (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/Discretizer.ipynb.h
useful for scikit-learn-like classifiers.

Reference

10.1 Classifier using Bayesian networks

class pyAgrum.skbn.**BNClassifier**(*learningMethod='GHC'*, *aPriori=None*, *scoringType='BIC'*, *constraints=None*, *aPrioriWeight=1*, *possibleSkeleton=None*, *DirichletCsv=None*, *discretizationStrategy='quantile'*, *discretizationNbBins=5*, *discretizationThreshold=25*, *usePR=False*, *significant_digit=10*)

Represents a (scikit-learn compliant) classifier wich uses a BN to classify. A BNClassifier is build using

- a Bayesian network,
- a database and a learning algorithm and parameters
- the use of BNDiscretizer to discretize with different algorithms some variables.

parameters:

learningMethod: str A string designating which type of learning we want to use. Possible values are: Chow-Liu, NaiveBayes, TAN, MIIC + (MDL ou NML), GHC, 3off2 + (MDL ou NML), Tabu. GHC designates Greedy Hill Climbing. MIIC designates Multivariate Information based Inductive Causation TAN designates Tree-augmented NaiveBayes Tabu designated Tabu list searching

aPriori: str A string designating the type of a priori smoothing we want to use. Possible values are Smoothing, BDeu, Dirichlet and NoPrior . Note: if using Dirichlet smoothing DirichletCsv cannot be set to none By default (when aPriori is None) : a smoothing(0.01) is applied.

scoringType: str A string designating the type of scoring we want to use. Since scoring is used while constructing the network and not when learning its parameters, the scoring will be ignored if using a learning algorithm with a fixed network structure such as Chow-Liu, TAN or NaiveBayes. possible values are: AIC, BIC, BD, BDeu, K2, Log2 AIC means Akaike information criterion BIC means Bayesian Information criterion BD means Bayesian-Dirichlet scoring BDeu means Bayesian-Dirichlet equivalent uniform Log2 means log2 likelihood ratio test

constraints: dict() A dictionary designating the constraints that we want to put on the structure of the Bayesian network. Ignored if using a learning algorithm where the structure is fixed such as TAN or NaiveBayes. the keys of the dictionary should be the strings "PossibleEdges" , "MandatoryArcs" and "ForbiddenArcs". The format of the values should be a tuple of strings (tail,head) which designates the string arc from tail to head. For example if we put the value ("x0"."y") in MandatoryArcs the network will surely have an arc going from x0 to y. Note: PossibleEdges allows for both (tail,head) and (head,tail) to be added to the Bayesian network, while the others are not symmetric.

aPrioriWeight: double The weight used for a priori smoothing.

possibleSkeleton: pyagrum.undigraph An undirected graph that serves as a possible skeleton for the Bayesian network

DirichletCsv: str the file name of the csv file we want to use for the dirichlet prior. Will be ignored if aPriori is not set to Dirichlet.

discretizationStrategy: str sets the default method of discretization for this discretizer. This method will be used if the user has not specified another method for that specific

variable using the setDiscretizationParameters method possible values are: 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'

defaultNumberOfBins: str or int sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. If the method used is NML, this parameter sets the the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

discretizationThreshold: int or float When using default parameters a variable will be treated as continous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For instance, if you have entered 0.95, the variable will be treated as continous only if more than 95% of its values are unique.

usePR: bool indicates if the threshold to choose is Prevision-Recall curve's threhsold or ROC's threshold by default. ROC curves should be used when there are roughly equal numbers of observations for each class. Precision-Recall curves should be used when there is a moderate to large class imbalance especially for the target's class.

significant_digit: number of significant digits when computing probabilities

XYfromCSV(*filename*, *with_labels=True*, *target=None*)

parameters:

filename: str the name of the csv file

with_labels: bool tells us whether the csv includes the labels themselves or their indexes.

target: str or None The name of the column that will be put in the dataframe y. If target is None, we use the target that is already specified in the classifier

returns:

X: pandas.dataframe Matrix containing the data

y: pandas.dataframe Column-vector containing the class for each data vector in X

Reads the data from a csv file and separates it into a X matrix and a y column vector.

fit(*X=None*, *y=None*, *filename=None*, *targetName=None*)

parameters:

X: {array-like, sparse matrix} of shape (n_samples, n_features) training data. Warning: Raises ValueError if either filename or targetname is not None. Raises ValueError if y is None.

y: array-like of shape (n_samples) Target values. Warning: Raises ValueError if either filename or targetname is not None. Raises ValueError if X is None

filename: str specifies the csv file where the training data and target values are located. Warning: Raises ValueError if either X or y is not None. Raises ValueError if targetName is None

targetName: str specifies the name of the targetVariable in the csv file. Warning: Raises ValueError if either X or y is not None. Raises ValueError if filename is None.

returns: void

Fits the model to the training data provided. The two possible uses of this function are fit(X,y) and fit(filename, targetName). Any other combination will raise a ValueError

fromTrainedModel(*bn*, *targetAttribute*, *targetModality=''*, *copy=False*, *threshold=0.5*, *variableList=None*)

parameters:

bn: pyagrum.BayesNet The Bayesian network we want to use for this classifier

targetAttribute: str the attribute that will be the target in this classifier

targetModality: str If this is a binary classifier we have to specify which modality we are looking at if the target attribute has more than 2 possible values if !="", a binary classifier is created. if ==””, a classifier is created that can be non binary depending on the number of modalities for targetAttribute. If binary, the second one is taken as targetModality.

copy: bool Indicates whether we want to put a copy of bn in the classifier, or bn itself.

threshold: double The classification threshold. If the probability that the target modality is true is larger than this threshold we predict that modality

variableList: list(str) A list of strings. variableList[i] is the name of the variable that has the index i. We use this information when calling predict to know which column corresponds to which variable. If this list is set to none, then we use the order in which the variables were added to the network.

returns: void

Creates a BN classifier from an already trained pyAgrum Bayesian network

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

predict(*X*, *with_labels=True*)

parameters:

X: {array-like, sparse matrix} of shape (n_samples, n_features) or str test data, can be either dataFrame, matrix or name of a csv file

with_labels: bool tells us whether the csv includes the labels themselves or their indexes.

returns:

y: array-like of shape (n_samples,) Predicted classes

Predicts the most likely class for each row of input data, with bn's Markov Blanket

predict_proba(*X*)

parameters:

X: {array-like, sparse matrix} of shape (n_samples, n_features) or str test data, can be either dataFrame, matrix or name of a csv file

returns:

y: array-like of shape (n_samples,) Predicted probability for each classes

Predicts the probability of classes for each row of input data, with bn's Markov Blanket

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.

- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – Mean accuracy of `self.predict(X)` wrt. *y*.

Return type float

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

showROC_PR(*filename*, *save_fig=False*, *show_progress=False*)
Use the *pyAgrum.lib.bn2roc* tools to create ROC and Precision-Recall curve

parameters:

csv_name [str] a csv filename

save_fig [bool] whether the graph soulb de saved

show_progress [bool] indicates if the resulting curve must be printed

10.2 Discretizer for Bayesian networks

class pyAgrum.skbn.**BNDiscretizer**(*defaultDiscretizationMethod='quantile'*, *defaultNumberOfBins=10*, *discretizationThreshold=25*)
Represents a tool to discretize some variables in a database in order to obtain a way to learn a pyAgrum's (discrete) Bayesian networks.

parameters:

defaultDiscretizationMethod: str sets the default method of discretization for this discretizer. Possible values are: 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'. This method will be used if the user has not specified another method for that specific variable using the setDiscretizationParameters method.

defaultNumberOfBins: str or int sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. If the method used is NML, this parameter sets the the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

discretizationThreshold: int or float When using default parameters a variable will be treated as continous only if it has more unique values than this number (if the number is an int greater than 1). If the number is a float between 0 and 1, we will test if the proportion of unique values is bigger than this number. For example if you have entered 0.95, the variable will be treated as continous only if more than 95% of its values are unique.

audit(*X*, *y=None*)

parameters:

X: {array-like, sparse matrix} of shape (n_samples, n_features) training data

y: array-like of shape (n_samples,) Target values

returns: auditDict: dict()

Audits the passed values of X and y. Tells us which columns in X we think are already discrete and which need to be discretized, as well as the discretization algorithm that will be used to discretize them The parameters which are suggested will be used when creating the variables. To change this the user can manually set discretization parameters for each variable using the setDiscretizationParameters function.

clear(*clearDiscretizationParameters=False*)

parameters:

clearDiscretizationParamaters: bool if True, this method also clears the parameters the user has set for each variable and resets them to the default.

returns: void

Sets the number of continous variables and the total number of bins created by this discretizer to 0. If clearDiscretizationParameters is True, also clears the the parameters for discretization the user has set for each variable.

createVariable(*variableName*, *X*, *y=None*, *possibleValuesY=None*)

parameters:

variableName: the name of the created variable

X: ndarray shape(n,1) A column vector containing n samples of a feature. The column for which the variable will be created

y: ndarray shape(n,1) A column vector containing the corresponding for each element in X.

possibleValuesX: onedimensional ndarray An ndarray containing all the unique values of X

possibleValuesY: onedimensional ndarray An ndarray containing all the unique values of y

returnModifiedX: bool X could be modified by this function during

returns:

var: pyagrum.DiscreteVariable the created variable

Creates a variable for the column passed in as a parameter and places it in the Bayesian network

discretizationCAIM(*x*, *y*, *possibleValuesX*, *possibleValuesY*)

parametres:

x: ndarray with shape (n,1) where n is the number of samples Column-vector that contains all the data that needs to be discretized

y: ndarray with shape (n,1) where n is the number of samples Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

possibleValuesY: one dimensional ndarray Contains the possible values of y. There should be two possible values since this is a binary classifier

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Applies the CAIM algorithm to discretize the values of x

discretizationElbowMethodRotation(*discretizationStrategy*, *X*)

parameters:

discretizationStrategy: str The method of discretization that will be used. Possible values are: 'quantile' , 'kmeans' and 'uniform'

X: one dimensional ndarray Contains the data that should be discretized

returns: binEdges: the edges of the bins the algorithm has chosen.

Calculates the sum of squared errors as a function of the number of clusters using the discretization strategy that is passed as a parameter. Returns the bins that are optimal for minimizing the variation and the number of bins at the same time. Uses the elbow method to find this optimal point. To find the "elbow" we rotate the curve and look for its minimum.

discretizationMDLP(*x*, *y*, *possibleValuesX*, *possibleValuesY*)

parametres:

x: ndarray with shape (n,1) where n is the number of samples Column-vector that contains all the data that needs to be discretized

y: ndarray with shape (n,1) where n is the number of samples Column-vector that contains the class for each sample. This vector will not be discretized, but the class-value of each sample is needed to properly apply the algorithm

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside

possibleValuesY: one dimensional ndarray Contains the possible values of y. There should be two possible values since this is a binary classifier

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Uses the MDLP algorithm described in Fayyad, 1995 to discretize the values of x.

discretizationNML(*X*, *possibleValuesX*, *kMax=10*, *epsilon=None*)

parameters:

X: one dimensional ndarray array that that contains all the data that needs to be discretized

possibleValuesX: one dimensional ndarray Contains all the possible values that x can take sorted in increasing order. There shouldn't be any doubles inside.

kMax: int the maximum number of bins before the algorithm stops itself.

epsilon: float or None the value of epsilon used in the algorithm. Should be as small as possible. If None is passed the value is automatically calculated.

returns: binEdges: a list of the edges of the bins that are chosen by this algorithm

Uses the disceretization algorithm described in "MDL Histogram Density Estimator", Kontkaken and Myllymaki, 2007 to discretize.

setDiscretizationParameters(*variableName=None*, *methode=None*, *numberOfBins=None*)

parameters:

variableName: str the name of the variable you want to set the discretization paramaters of. Set to None to set the new default for this BNClassifier.

methode: str The method of discretization used for this variable. Type "NoDiscretization" if you do not want to discretize this variable. Possible values are: 'NoDiscretization', 'quantile', 'uniform', 'kmeans', 'NML', 'CAIM' and 'MDLP'

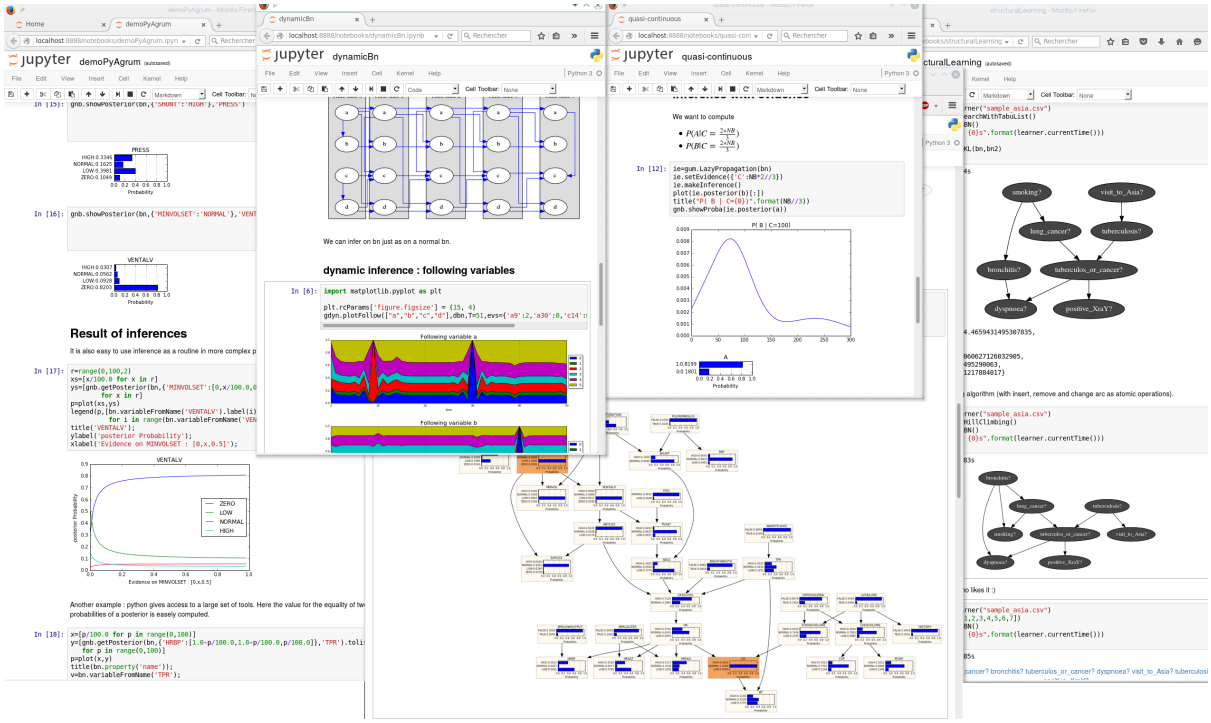
numberOfBins: sets the number of bins if the method used is quantile, kmeans, uniform. In this case this parameter can also be set to the string 'elbowMethod' so that the best number of bins is found automatically. if the method used is NML, this parameter sets the the maximum number of bins up to which the NML algorithm searches for the optimal number of bins. In this case this parameter must be an int If any other discetization method is used, this parameter is ignored.

returns: void

CHAPTER 11

pyAgrum.lib.notebook

`pyAgrum.lib.notebook` aims to facilitate the use of pyAgrum with jupyter notebook (or lab).

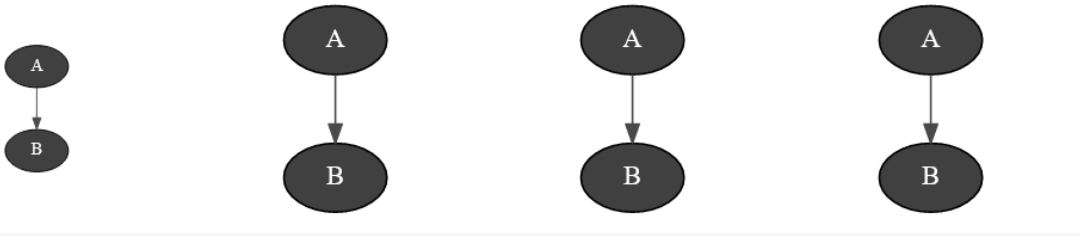


11.1 Visualization of graphical models

Important: For many graphical representations functions, the parameter *size* is directly transferred to *graphviz*. Hence, Its format is a string containing an int. However if *size* ends in an exclamation point "!" (such as *size="4!"*), then *size* is taken to be the desired minimum size. In this case, if both dimensions of the drawing are less than size, the drawing is scaled up uniformly until at least one dimension equals its dimension in size.

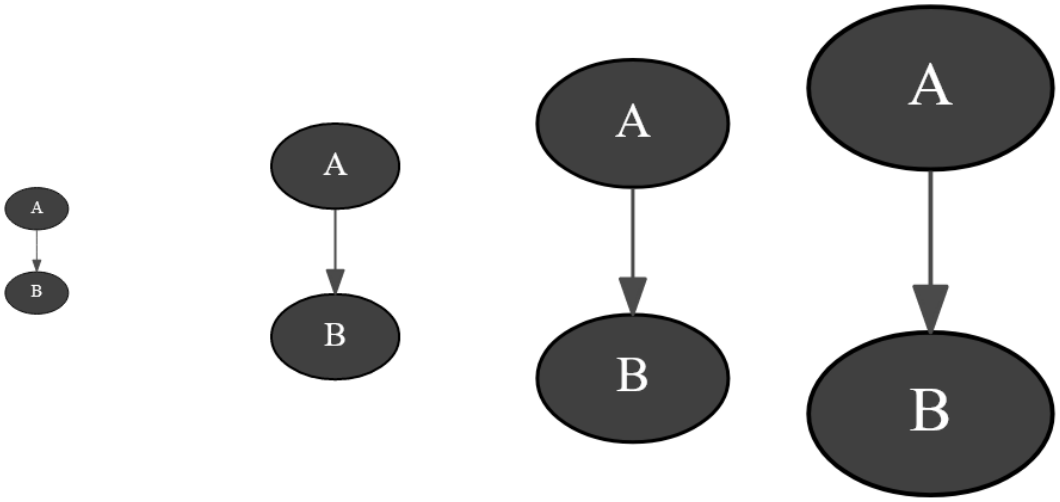
```
bn=gum.fastBN("A->B")
print("* withtout '!'")
gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}") for i in range(1,5)],captions=[f'size="{i}"' for i in range(1,5)])

print("* witht '!'")
gnb.sideBySide(*[gnb.getBN(bn,size=f"{i}!") for i in range(1,5)],captions=[f'size="{i}!"' for i in range(1,5)])
```

```
* withtout '!'
```



size="1" size="2" size="3" size="4"

```
* witht '!'
```



size="1!" size="2!" size="3!" size="4!"

pyAgrum.lib.notebook.**showBN**(*bn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

show a Bayesian network

Parameters

- **bn** – the Bayesian network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**getBN**(*bn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

get a HTML string for a Bayesian network

Parameters

- **bn** – the Bayesian network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**showInfluenceDiagram**(*diag*, *size=None*)
show an influence diagram as a graph

Parameters

- **diag** – the influence diagram
- **size** – size of the rendered graph

Returns the representation of the influence diagram

pyAgrum.lib.notebook.**getInfluenceDiagram**(*diag*, *size=None*)
get a HTML string for an influence diagram as a graph

Parameters

- **diag** – the influence diagram
- **size** – size of the rendered graph

Returns the HTML representation of the influence diagram

pyAgrum.lib.notebook.**showMN**(*mn*, *view=None*, *size=None*, *nodeColor=None*, *factorColor=None*, *edgeWidth=None*, *edgeColor=None*, *cmap=None*, *cmapEdge=None*)
show a Markov network

Parameters

- **mn** – the markov network
- **view** – 'graph' | 'factorgraph' | None (default)
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a function returning a value (beeween 0 and 1) to be shown as a color of factor. (used when view='factorgraph')
- **edgeWidth** – a edgeMap of values to be shown as width of edges (used when view='graph')
- **edgeColor** – a edgeMap of values (between 0 and 1) to be shown as color of edges (used when view='graph')
- **cmap** – color map to show the colors
- **cmapEdge** – color map to show the edge color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**getMN**(*mn*, *view=None*, *size=None*, *nodeColor=None*, *factorColor=None*, *edgeWidth=None*, *edgeColor=None*, *cmap=None*, *cmapEdge=None*)

get an HTML string for a Markov network

Parameters

- **mn** – the markov network
- **view** – 'graph' | 'factorgraph' | None (default)
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a function returning a value (beeween 0 and 1) to be shown as a color of factor. (used when view='factorgraph')
- **edgeWidth** – a edgeMap of values to be shown as width of edges (used when view='graph')
- **edgeColor** – a edgeMap of values (between 0 and 1) to be shown as color of edges (used when view='graph')
- **cmap** – color map to show the colors
- **cmapEdge** – color map to show the edge color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**showCN**(*cn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

show a credal network

Parameters

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**getCN**(*cn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmap=None*, *cmapArc=None*)

get a HTML string for a credal network

Parameters

- **cn** – the credal network
- **size** – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the colors
- **cmapArc** – color map to show the arc color if distinction is needed

Returns the graph

pyAgrum.lib.notebook.**showInference**(*model*, ***kwargs*)
show pydot graph for an inference in a notebook

Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

pyAgrum.lib.notebook.**getInference**(*model*, ***kwargs*)
get a HTML string for an inference in a notebook

Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)

- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

pyAgrum.lib.notebook.**showJunctionTree**(*bn*, *withNames=True*, *size=None*)
Show a junction tree

Parameters

- **bn** – the Bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

Returns the representation of the graph

pyAgrum.lib.notebook.**getJunctionTree**(*bn*, *withNames=True*, *size=None*)
get a HTML string for a junction tree (more specifically a join tree)

Parameters

- **bn** – the Bayesian network
- **withNames** (*boolean*) – display the variable names or the node id in the clique
- **size** – size of the rendered graph

Returns the HTML representation of the graph

11.2 Visualization of Potentials

pyAgrum.lib.notebook.**showProba**(*p*, *scale=1.0*)
Show a mono-dim Potential

Parameters

- **p** – the mono-dim Potential
- **scale** – the scale (zoom)

pyAgrum.lib.notebook.**getPosterior**(*bn*, *evs*, *target*)
shortcut for proba2histo(gum.getPosterior(bn,evs,target))

Parameters

- **bn** (*gum.BayesNet*) – the BayesNet
- **evs** (*dict(str->int)*) – map of evidence
- **target** (*str*) – name of target variable

Returns the matplotlib graph

pyAgrum.lib.notebook.**showPosterior**(*bn*, *evs*, *target*)
shortcut for showProba(gum.getPosterior(bn,evs,target))

Parameters

- **bn** – the BayesNet
- **evs** – map of evidence
- **target** – name of target variable

pyAgrum.lib.notebook.**getPotential**(*pot*, *digits=None*, *withColors=None*, *varnames=None*)
return a HTML string of a gum.Potential as a HTML table. The first dimension is special (horizontal) due to the representation of conditional probability table

Parameters

- **pot** (*gum.Potential*) – the potential to get
- **digits** (*int*) – number of digits to show
- **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

11.3 Exporting visualisations (as pdf,png)

pyAgrum.lib.notebook.**export**(*model*, *filename*, ***kwargs*)

export the graphical representation of the model in filename (png, pdf,etc.)

Parameters

- **model** (*GraphicalModel*) – the model to show (pyAgrum.BayesNet, pyAgrum.MarkovNet, pyAgrum.InfluenceDiagram or pyAgrum.Potential)
- **filename** (*str*) – the name of the resulting file (suffix in ['pdf', 'png', 'fig', 'jpg', 'svg', 'ps'])

pyAgrum.lib.notebook.**exportInference**(*model*, *filename*, ***kwargs*)

the graphical representation of an inference in a notebook

Parameters

- **model** (*GraphicalModel*) – the model in which to infer (pyAgrum.BayesNet, pyAgrum.MarkovNet or pyAgrum.InfluenceDiagram)
- **filename** (*str*) – the name of the resulting file (suffix in ['pdf', 'png', 'ps'])
- **engine** (*gum.Inference*) – inference algorithm used. If None, gum.LazyPropagation will be used for BayesNet, gum.ShaferShenoy for gum.MarkovNet and gum.ShaferShenoyLIMIDInference for gum.InfluenceDiagram.
- **evs** (*dictionnary*) – map of evidence
- **targets** (*set*) – set of targets
- **size** (*string*) – size of the rendered graph
- **nodeColor** – a nodeMap of values (between 0 and 1) to be shown as color of nodes (with special colors for 0 and 1)
- **factorColor** – a nodeMap of values (between 0 and 1) to be shown as color of factors (in MarkovNet representation)
- **arcWidth** – a arcMap of values to be shown as width of arcs
- **arcColor** – a arcMap of values (between 0 and 1) to be shown as color of arcs

- **cmap** – color map to show the color of nodes and arcs
- **cmapArc** – color map to show the vals of Arcs.
- **graph** – only shows nodes that have their id in the graph (and not in the whole BN)
- **view** – graph | factorgraph | None (default) for Markov network

Returns the desired representation of the inference

11.4 Visualization of graphs

pyAgrum.lib.notebook.**getDot**(*dotstring*, *size=None*)
get a dot string as a HTML string

Parameters

- **dotstring** – dot string
- **size** – size of the rendered graph
- **format** – render as "png" or "svg"
- **bg** – color for background

Returns the HTML representation of the graph

pyAgrum.lib.notebook.**showDot**(*dotstring*, *size=None*)
show a dot string as a graph

Parameters

- **dotstring** – dot string
- **size** – size of the rendered graph

Returns the representation of the graph

pyAgrum.lib.notebook.**getGraph**(*gr*, *size=None*)
get a HTML string representation of pydot graph

Parameters

- **gr** – pydot graph
- **size** – size of the rendered graph
- **format** – render as "png" or "svg"

Returns the HTML representation of the graph as a string

pyAgrum.lib.notebook.**showGraph**(*gr*, *size=None*)
show a pydot graph in a notebook

Parameters

- **gr** – pydot graph
- **size** – size of the rendered graph

Returns the representation of the graph

11.5 Visualization of approximation algorithm

pyAgrum.lib.notebook.**animApproximationScheme**(*apsc*, *scale=<ufunc 'log10'>*)
show an animated version of an approximation algorithm

Parameters

- **apsc** – the approximation algorithm
- **scale** – a function to apply to the figure

11.6 Helpers

pyAgrum.lib.notebook.**configuration**()
Display the collection of dependance and versions

pyAgrum.lib.notebook.**sideBySide**(**args*, ***kwargs*)
display side by side args as HMTL fragment (using string, _repr_html_() or str())

Parameters

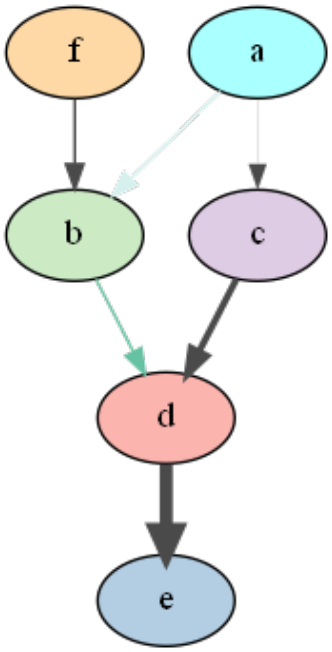
- **args** – HMTL fragments as string arg, arg._repr_html_() or str(arg)
- **captions** – list of strings (captions)

CHAPTER 12

Module bn2graph

A module to graphically display Bayesian networks using pydotplus (https://pypi.org/project/pydotplus/) (and then graphviz (https://graphviz.org/)).

```
import pyAgrum as gum
from pyAgrum.lib.bn2graph import BN2dot

bn = gum.fastBN("a->b->d;a->c->d[3]->e;f->b")
g = BN2dot(bn,
           nodeColor={'a': 1,
                      'b': 0.3,
                      'c': 0.4,
                      'd': 0.1,
                      'e': 0.2,
                      'f': 0.5},
           arcColor={(0, 1): 0.2,
                     (1, 2): 0.5},
           arcWidth={(0, 3): 0.4,
                     (3, 2): 0.5,
                     (2,4) :0.6})

g.write("bn2graph_test.png", format='png')
```



12.1 Visualization of Potentials

12.2 Visualization of Bayesian networks

pyAgrum.lib.bn2graph.**BN2dot**(*bn*, *size=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmapNode=None*, *cmapArc=None*, *showMsg=None*)

create a pydotplus representation of the BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the Bayesian network
- **size** (*str*) – size of the rendered graph
- **nodeColor** (*dict*) – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
- **arcWidth** (*dict*) – a arcMap of values to be shown as bold arcs
- **arcColor** (*dict*) – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** (*ColorMap*) – color map to show the vals of Nodes
- **cmapArc** (*ColorMap*) – color map to show the vals of Arcs
- **showMsg** (*dict*) – a nodeMap of values to be shown as tooltip

Returns

Return type the desired representation of the Bayesian network

pyAgrum.lib.bn2graph.**BNinference2dot**(*bn*, *size=None*, *engine=None*, *evs=None*, *targets=None*, *nodeColor=None*, *arcWidth=None*, *arcColor=None*, *cmapNode=None*, *cmapArc=None*, *dag=None*)

create a pydotplus representation of an inference in a BN

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – the Bayesian network
- **size** (*str*) – size of the rendered graph
- **engine** (*pyAgrum.Inference*) – inference algorithm used. If None, LazyPropagation will be used
- **evs** (*dict*) – map of evidence
- **targets** (*set*) – set of targets. If targets={} then each node is a target
- **nodeColor** (*dict*) – a nodeMap of values to be shown as color nodes (with special color for 0 and 1)
- **arcWidth** (*dict*) – a arcMap of values to be shown as bold arcs
- **arcColor** (*dict*) – a arcMap of values (between 0 and 1) to be shown as color of arcs
- **cmapNode** (*ColorMap*) – color map to show the vals of Nodes
- **cmapArc** (*ColorMap*) – color map to show the vals of Arcs
- **dag** (pyAgrum.DAG (page 7)) – only shows nodes that have their id in the dag (and not in the whole BN)

Returns

Return type the desired representation of the inference

12.3 Hi-level functions

CHAPTER 13

pyAgrum.lib.explain

The purpose of `pyAgrum.lib.explain` is to give tools to explain and interpret the structure and parameters of a Bayesian network.

13.1 Dealing with independence

pyAgrum.lib.explain.**independenceListForPairs**(*bn*, *filename*, *target=None*, *plot=True*, *alphabetic=False*)

get the p-values of the chi2 test of a (as simple as possible) independence proposition for every non arc.

Parameters

- **bn** (*gum.BayesNet*) – the Bayesian network
- **filename** (*str*) – the name of the csv database
- **alphabetic** (*bool*) – if True, the list is alphabetically sorted else it is sorted by the p-value
- **target** (*(optional) str or int*) – the name or id of the target variable
- **plot** (*bool*) – if True, plot the result

Returns

Return type the list

13.2 Dealing with mutual information and entropy

pyAgrum.lib.explain.**getInformation**(*bn*, *evs=None*, *size=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*)

get a HTML string for a bn annotated with results from inference : entropy and mutual information

Parameters

- **bn** – the BN
- **evs** – map of evidence

- **size** – size of the graph
- **cmap** – colour map used

Returns the HTML string

pyAgrum.lib.explain.**showInformation**(*bn*, *evs=None*, *target=None*, *size=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*)

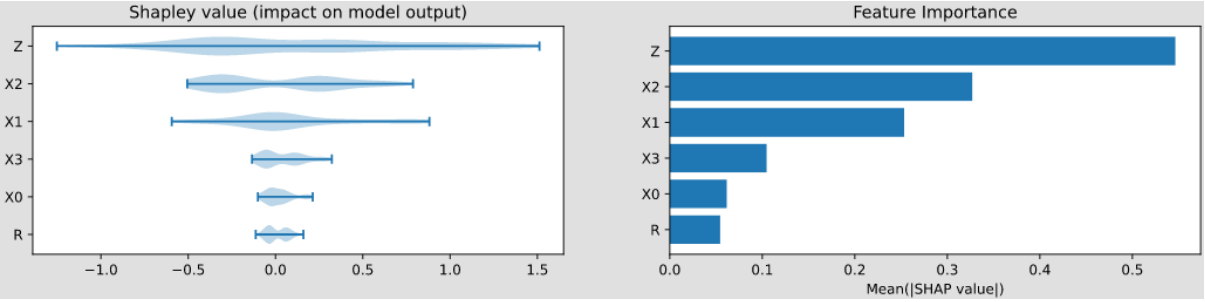
show a bn annotated with results from inference : entropy and mutual information

Parameters

- **bn** – the BN
- **evs** – map of evidence
- **target** – (optional) the name or id of the target variable
- **size** – size of the graph
- **cmap** – colour map used

Returns the graph

13.3 Dealing with ShapValues



class pyAgrum.lib.explain.**ShapValues**(*bn*, *target*)

Bases: object

The ShapValue class implements the calculation of Shap values in Bayesian networks.

The main implementation is based on Conditional Shap values[3], but the Interventional calculation method proposed in[2] is also present. In addition, a new causal method, based on[1], is implemented which is well suited for Bayesian networks.

bn [gum.BayesNet] The Bayesian network

target [str] the name of the target node

causal(*train*, *plot=False*, *plot_importance=False*, *percentage=False*)

Compute the causal Shap Values for each variables.

Parameters

- **train** (*pandas.DataFrame*) – the database
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot_importance** (*bool*) – if True, plot the importance plot

[3] Lundberg, S. M., Su-In, L. (2017). A Unified Approach to Interpreting Model. 31st Conference on Neural Information Processing Systems. Long Beach, CA, USA.

[2] Janzing, D., Minorics, L., Blöbaum, P. (2019). Feature relevance quantification in explainable AI: A causality problem. arXiv: Machine Learning. Retrieved 6 24, 2021, from https://arxiv.org/abs/1910.13413

[1] Heskes, T., Sijben, E., Bucur, I., Claassen, T. (2020). Causal Shapley Values: Exploiting Causal Knowledge. 34th Conference on Neural Information Processing Systems. Vancouver, Canada.

- **percentage** (*bool*) – if True, the importance plot is shown in percent.

Returns

Return type a dictionary Dict[str,float]

conditional(*train*, *plot=False*, *plot_importance=False*, *percentage=False*)
Compute the conditional Shap Values for each variables.

Parameters

- **train** (*pandas.DataFrame*) – the database
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

Returns

Return type a dictionary Dict[str,float]

marginal(*train*, *sample_size=200*, *plot=False*, *plot_importance=False*, *percentage=False*)
Compute the marginal Shap Values for each variables.

Parameters

- **train** (*pandas.DataFrame*) – the database
- **sample_size** (*int*) – The computation of marginal ShapValue is very slow. The parameter allow to compute only on a fragment of the database.
- **plot** (*bool*) – if True, plot the violin graph of the shap values
- **plot_importance** (*bool*) – if True, plot the importance plot
- **percentage** (*bool*) – if True, the importance plot is shown in percent.

Returns

Return type a dictionary Dict[str,float]

showShapValues(*results*, *cmap='plasma'*)

Parameters

- **results** (*dict[str,float]*) – The (Shap) values associates to each variable
- **cmap** (*Matplotlib.ColorMap*) – The colormap used for colouring the nodes

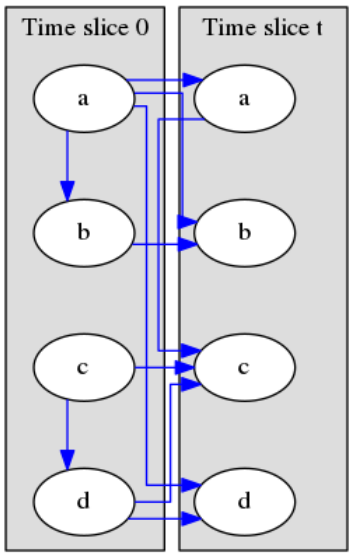
Returns

Return type a pydotplus.graph

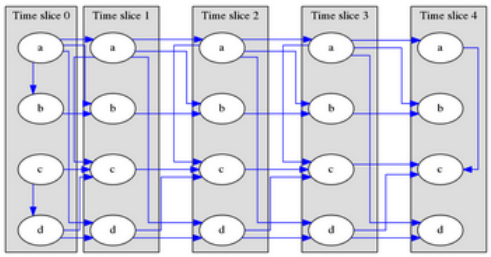
CHAPTER 14

Module dynamic Bayesian network

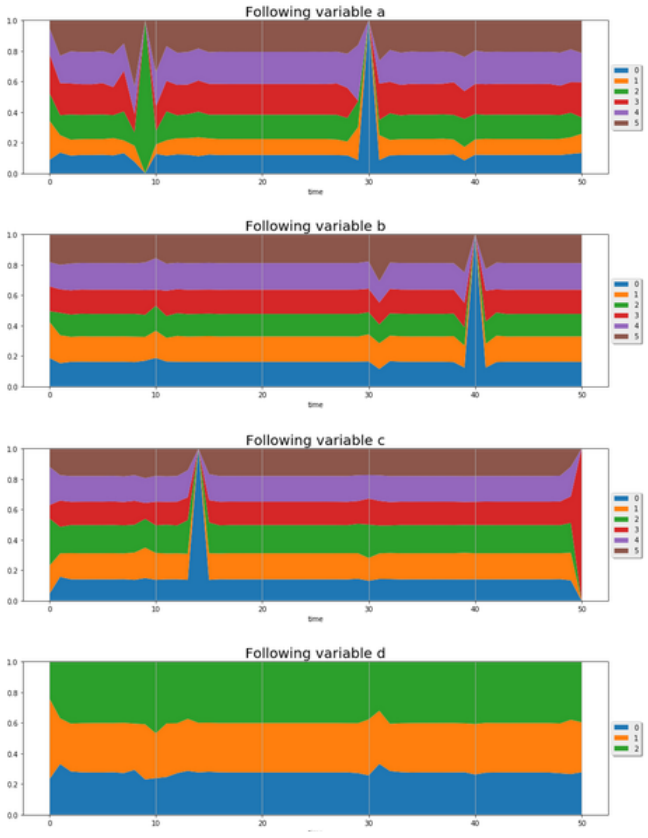
```
gdyn.showTimeSlices(dbn,format="png")
```




```
T=5

bn=gdyn.unroll2TBN(dbn,T)
gdyn.showTimeSlices(bn,size="10")
```




```
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (15, 4)
gdyn.plotFollow(["a","b","c","d"],dbn,T=51,evs={'a9':2,'a
30':0,'c14':0,'b40':0,'c50':3})
```



The purpose of this module is to provide basic tools for dealing with dynamic Bayesian Network (and inference) : modeling, visualisation, inference.

pyAgrum.lib.dynamicBN.**getTimeSlices**(*dbn*, *size=None*)

Try to correctly represent dBN and 2TBN as an HTML string

Parameters

- **dbn** – the dynamic BN
- **size** – size of the fig

pyAgrum.lib.dynamicBN.**getTimeSlicesRange**(*dbn*)
get the range and (name,radical) of each variables

Parameters **dbn** – a 2TBN or an unrolled BN

Returns all the timeslice of a dbn

e.g. ['0','t'] for a classic 2TBN range(T) for a classic unrolled BN

pyAgrum.lib.dynamicBN.**is2TBN**(*bn*)
Check if bn is a 2 TimeSlice Bayesian network

Parameters **bn** – the Bayesian network

Returns True if the BN is syntaxically correct to be a 2TBN

pyAgrum.lib.dynamicBN.**plotFollow**(*lovars*, *twoTdbn*, *T*, *evs*)
plots modifications of variables in a 2TDN knowing the size of the time window (T) and the evidence on the sequence.

Parameters

- **lovars** – list of variables to follow
- **twoTdbn** – the two-timeslice dbn
- **T** – the time range
- **evs** – observations

pyAgrum.lib.dynamicBN.**plotFollowUnrolled**(*lovars*, *dbn*, *T*, *evs*)
plot the dynamic evolution of a list of vars with a dBN

Parameters

- **lovars** – list of variables to follow
- **dbn** – the unrolled dbn
- **T** – the time range
- **evs** – observations

pyAgrum.lib.dynamicBN.**realNameFrom2TBNname**(*name*, *ts*)
@return dynamic name from static name and timeslice (no check)

pyAgrum.lib.dynamicBN.**showTimeSlices**(*dbn*, *size=None*)
Try to correctly display dBN and 2TBN

Parameters

- **dbn** – the dynamic BN
- **size** – size of the fig

pyAgrum.lib.dynamicBN.**unroll2TBN**(*dbn*, *nbr*)
unroll a 2TBN given the nbr of timeslices

Parameters

- **dbn** – the dBN
- **nbr** – the number of timeslice

Returns unrolled BN from a 2TBN and the nbr of timeslices

CHAPTER 15

other pyAgrum.lib modules

15.1 bn2roc

The purpose of this module is to provide tools for building ROC and PR from Bayesian Network.

pyAgrum.lib.bn2roc.**showPR**(*bn*, *csv_name*, *target*, *label*, *show_progress=True*, *show_fig=True*, *save_fig=False*, *with_labels=True*, *significant_digits=10*)

Compute the ROC curve and save the result in the folder of the csv file.

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – a Bayesian network
- **csv_name** (*str*) – a csv filename
- **target** (*str*) – the target
- **label** (*str*) – the target label
- **show_progress** (*bool*) – indicates if the progress bar must be printed
- **save_fig** – save the result ?
- **show_fig** – plot the resuls ?
- **with_labels** – labels in csv ?
- **significant_digits** – number of significant digits when computing probabilities

pyAgrum.lib.bn2roc.**showROC**(*bn*, *csv_name*, *target*, *label*, *show_progress=True*, *show_fig=True*, *save_fig=False*, *with_labels=True*, *significant_digits=10*)

Compute the ROC curve and save the result in the folder of the csv file.

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – a Bayesian network
- **csv_name** (*str*) – a csv filename
- **target** (*str*) – the target
- **label** (*str*) – the target label
- **show_progress** (*bool*) – indicates if the progress bar must be printed
- **save_fig** – save the result

- **show_fig** – plot the resuls
- **with_labels** – labels in csv
- **significant_digits** – number of significant digits when computing probabilities

pyAgrum.lib.bn2roc.**showROC_PR**(*bn*, *csv_name*, *target*, *label*, *show_progress=True*, *show_fig=True*, *save_fig=False*, *with_labels=True*, *show_ROC=True*, *show_PR=True*, *significant_digits=10*)

Compute the ROC curve and save the result in the folder of the csv file.

Parameters

- **bn** (pyAgrum.BayesNet (page 48)) – a Bayesian network
- **csv_name** (*str*) – a csv filename
- **target** (*str*) – the target
- **label** (*str*) – the target label
- **show_progress** (*bool*) – indicates if the progress bar must be printed
- **save_fig** – save the result
- **show_fig** – plot the resuls
- **with_labels** – labels in csv
- **show_ROC** (*bool*) – whether we show the ROC figure
- **show_PR** (*bool*) – whether we show the PR figure
- **significant_digits** – number of significant digits when computing probabilitie

Returns (pointsROC, thresholdROC, pointsPR, thresholdPR)

Return type tuple

15.2 bn2scores

The purpose of this module is to provide tools for computing different scores from a BN.

pyAgrum.lib.bn2scores.**checkCompatibility**(*bn*, *fields*, *csv_name*)

check if the variables of the bn are in the fields

Parameters

- **bn** – gum.BayesNet
- **fields** – Dict of name,position in the file
- **csv_name** – name of the csv file

@throw gum.DatabaseError if a BN variable is not in fields

Returns return a dictionary of position for BN variables in fields

pyAgrum.lib.bn2scores.**computeScores**(*bn_name*, *csv_name*, *visible=False*, *transforme_label=False*)

Compute scores from a bn w.r.t to a csv :param bn_name: a gum.BayesianNetwork or a filename for a BN :param csv_name: a filename for the CSV database :param visible: do we show the progress :param transforme_label: do we adapt from labels to id :return: percentDatabaseUsed,scores

pyAgrum.lib.bn2scores.**lines_count**(*filename*)

count lines in a file

15.3 bn_vs_bn

The purpose of this module is to provide tools for comaring different BNs.

class pyAgrum.lib.bn_vs_bn.**GraphicalBNComparator**(*name1*, *name2*, *delta=1e-06*)
Bases: object

BNGraphicalComparator allows to compare in multiple way 2 BNs... The smallest assumption is that the names of the variables are the same in the 2 BNs. But some comparisons will have also to check the type and domainSize of the variables. The bns have not exactly the same role : _bn1 is rather the referent model for the comparison whereas _bn2 is the compared one to the referent model.

Parameters

- **name1** (*str or* pyAgrum.BayesNet (page 48)) – a BN or a filename for reference
- **name2** (*str or* pyAgrum.BayesNet (page 48)) – another BN or antoher filename for comparison

dotDiff()
Return a pydotplus graph that compares the arcs of _bn1 (reference) with those of self._bn2. full black line: the arc is common for both full red line: the arc is common but inverted in _bn2 dotted black line: the arc is added in _bn2 dotted red line: the arc is removed in _bn2

Warning: if pydotplus is not installed, this function just returns None

Returns the result dot graph or None if pydotplus can not be imported

Return type pydotplus.Dot

equivalentBNs()
Check if the 2 BNs are equivalent :

- same variables
- same graphical structure
- same parameters

Returns "OK" if bn are the same, a description of the error otherwise

Return type str

hamming()
Compute hamming and structural hamming distance

Hamming distance is the difference of edges comparing the 2 skeletons, and Structural Hamming difference is the difference comparing the cpdags, including the arcs' orientation.

Returns A dictionnary containing 'hamming','structural hamming'

Return type dict[double,double]

scores()
Compute Precision, Recall, F-score for self._bn2 compared to self._bn1

precision and recall are computed considering BN1 as the reference

Fscor is 2*(recall* precision)/(recall+precision) and is the weighted average of Precision and Recall.

dist2opt=square root of (1-precision)^2+(1-recall)^2 and represents the euclidian distance to the ideal point (precision=1, recall=1)

Returns A dictionnary containing 'precision', 'recall', 'fscore', 'dist2opt' and so on.

Return type dict[str,double]

skeletonScores()

Compute Precision, Recall, F-score for skeletons of self._bn2 compared to self._bn1

precision and recall are computed considering BN1 as the reference

Fscor is 2*(recall* precision)/(recall+precision) and is the weighted average of Precision and Recall.

dist2opt=square root of (1-precision)^2+(1-recall)^2 and represents the euclidian distance to the ideal point (precision=1, recall=1)

Returns A dictionnary containing 'precision', 'recall', 'fscore', 'dist2opt' and so on.

Return type dict[str,double]

CHAPTER 16

Functions from pyAgrum

16.1 Useful functions in pyAgrum

pyAgrum.**about**()
about() for pyAgrum

pyAgrum.**getPosterior**(*model*, *evs*, *target*)
Compute the posterior of a single target (variable) in a BN given evidence

getPosterior uses a VariableElimination inference. If more than one target is needed with the same set of evidence or if the same target is needed with more than one set of evidence, this function is not relevant since it creates a new inference engine every time it is called.

Parameters

- **bn** (pyAgrum.BayesNet (page 48) *or* pyAgrum.MarkovNet (page 158)) – The probabilistic Graphical Model
- **evs** (*dictionaryDict*) – {name/id:val, name/id : [val1, val2], ... }
- **target** (*string or int*) – variable name or id

Returns

Return type posterior (*pyAgrum.Potential* (page 39) or other)

16.2 Quick specification of (randomly parameterized) graphical models

aGrUM/pyAgrum offers a compact syntax that allows to quickly specify prototypes of graphical models. These *fastPrototype* aGrUM's methods have also been wrapped in functions of pyAgrum.

```
gum.fastBN("A->B<-C;B->D")
```

The type of the random variables can be specifiy with different syntaxes:

- by default, a variable is a *pyAgrum.RangeVariable* (page 28) using the default domain size (second argument of the functions).

- with `a[10]`, the variable is a *pyAgrum.RangeVariable* (page 28) using 10 as domain size (from 0 to 9)
- with `a[3,7]`, the variable is a *pyAgrum.RangeVariable* (page 28) using a domainSize from 3 to 7
- with `a[1,3.14,5,6.2]`, the variable is a *pyAgrum.DiscretizedVariable* (page 26) using the given ticks (at least 3 values)
- with `a{top|middle|bottom}`, the variable is a *pyAgrum.LabelizedVariable* (page 23) using the given labels (here : 'top', 'middle' and 'bottom').
- with `a{-1|5|0|3}`, the variable is a `pyAgrum.IntegerVariable` using the sorted given values.

Note:

- If the dot-like string contains such a specification more than once for a variable, the first specification will be used.
- the CPTs are randomly generated.

---

pyAgrum.**fastBN**(*structure*, *domain_size=2*)

Create a Bayesian network with a dot-like syntax which specifies:

- the structure 'a->b->c;b->d<-e;',
- the type of the variables with different syntax (cf documentation).

Examples

```
>>> import pyAgrum as gum
>>> bn=gum.fastBN('A->B[1,3]<-C{yes|No}->D[2,4]<-E[1,2.5,3.9]',6)
```

Parameters

- **structure** (*str*) – the string containing the specification
- **domain_size** (*int*) – the default domain size for variables

Returns the resulting bayesian network

Return type *pyAgrum.BayesNet* (page 48)

pyAgrum.**fastMN**(*structure*, *domain_size=2*)

Create a Markov network with a modified dot-like syntax which specifies:

- the structure 'a-b-c;b-d;c-e;' where each chain 'a-b-c' specifies a factor,
- the type of the variables with different syntax (cf documentation).

Examples

```
>>> import pyAgrum as gum
>>> bn=gum.fastMN('A--B[1,3]--C{yes|No};C--D[2,4]--E[1,2.5,3.9]',6)
```

Parameters

- **structure** (*str*) – the string containing the specification
- **domain_size** (*int*) – the default domain size for variables

Returns the resulting Markov network

Return type *pyAgrum.MarkovNet* (page 158)

pyAgrum.**fastID**(*structure*, *domain_size=2*)

Create an Influence Diagram with a modified dot-like syntax which specifies:

- the structure 'a->b<-c;b->d;c<-e;',
- the type of the variables with different syntax (cf documentation),
- a prefix for the type of node (chance/decision/utiliy nodes):
  - `a` : a chance node named 'a' (by default)
  - `$a` : a utility node named 'a'
  - `*a` : a decision node named 'a'

Examples

```
>>> import pyAgrum as gum
>>> bn=gum.fastID('A->B[1,3]<-*C{yes|No}->$D<-E[1,2.5,3.9]',6)
```

Parameters

- **structure** (*str*) – the string containing the specification
- **domain_size** (*int*) – the default domain size for variables

Returns the resulting Influence Diagram

Return type *pyAgrum.InfluenceDiagram* (page 170)

16.3 Input/Output for Bayesian networks

pyAgrum.**availableBNExts**()

Give the list of all formats known by pyAgrum to save a Bayesian network.

Returns a string which lists all suffixes for supported BN file formats.

pyAgrum.**loadBN**(*filename*, *listeners=None*, *verbose=False*, ***opts*)

load a BN from a file with optional listeners and arguments

Parameters

- **filename** – the name of the input file
- **listeners** – list of functions to execute
- **verbose** – whether to print or not warning messages
- **system** – (for O3PRM) name of the system to flatten in a BN
- **classpath** – (for O3PRM) list of folders containing classes

Returns a BN from a file using one of the availableBNExts() suffixes.

Listeners could be added in order to monitor its loading.

Examples

```
>>> import pyAgrum as gum
>>>
>>> # creating listeners
>>> def foo_listener(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadBN('./bn.bif',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

pyAgrum.**saveBN**(*bn*, *filename*)
save a BN into a file using the format corresponding to one of the availableWriteBNExts() suffixes.

Parameters

- **bn(gum.BayesNet)** – the BN to save
- **filename(str)** – the name of the output file

16.4 Input/Output for Markov networks

pyAgrum.**availableMNExts**()
Give the list of all formats known by pyAgrum to save a Markov network.

Returns a string which lists all suffixes for supported MN file formats.

pyAgrum.**loadMN**(*filename*, *listeners=None*, *verbose=False*)
load a MN from a file with optional listeners and arguments

Parameters

- **filename** – the name of the input file
- **listeners** – list of functions to execute
- **verbose** – whether to print or not warning messages

Returns a MN from a file using one of the availableMNExts() suffixes.

Listeners could be added in order to monitor its loading.

Examples

```
>>> import pyAgrum as gum
>>>
>>> # creating listeners
>>> def foo_listener(progress):
>>>     if progress==200:
```

(continues on next page)

(continued from previous page)

```
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar_listener(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>> # loadBN with list of listeners
>>> gum.loadMN('./bn.uai',listeners=[foo_listener,bar_listener])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

pyAgrum.**saveMN**(*mn*, *filename*)
save a MN into a file using the format corresponding to one of the availableWriteMNExts() suffixes.

Parameters

- **mn(gum.MarkovNet)** – the MN to save
- **filename(str)** – the name of the output file

16.5 Input for influence diagram

pyAgrum.**availableIDExts**()
Give the list of all formats known by pyAgrum to save a influence diagram.

Returns a string which lists all suffixes for supported ID file formats.

pyAgrum.**loadID**(*filename*)
read a gum.InfluenceDiagram from a ID file

Parameters **filename** – the name of the input file

Returns an InfluenceDiagram

pyAgrum.**saveID**(*infdiag*, *filename*)
save an ID into a file using the format corresponding to one of the availableWriteIDExts() suffixes.

Parameters

- **ID(gum.InfluenceDiagram)** – the ID to save
- **filename(str)** – the name of the output file

CHAPTER 17

Other functions from aGrUM

17.1 Listeners

aGrUM includes a mechanism for listening to actions (close to QT signal/slot). Some of them have been ported to pyAgrum :

17.1.1 LoadListener

Listeners could be added in order to monitor the progress when loading a pyAgrum.BayesNet

```
>>> import pyAgrum as gum
>>>
>>> # creating a new liseners
>>> def foo(progress):
>>>     if progress==200:
>>>         print(' BN loaded ')
>>>         return
>>>     elif progress==100:
>>>         car='%'
>>>     elif progress%10==0:
>>>         car='#'
>>>     else:
>>>         car='.'
>>>     print(car,end='',flush=True)
>>>
>>> def bar(progress):
>>>     if progress==50:
>>>         print('50%')
>>>
>>>
>>> gum.loadBN('./bn.bif',listeners=[foo,bar])
>>> # .........#.........#.........#.........#..50%
>>> # .......#.........#.........#.........#.........#.........% | bn loaded
```

17.1.2 StructuralListener

Listeners could also be added when structural modification are made in a pyAgrum.BayesNet:

```
>>> import pyAgrum as gum
>>>
>>> ## creating a BayesNet
>>> bn=gum.BayesNet()
>>>
>>> ## adding structural listeners
>>> bn.addStructureListener(whenNodeAdded=lambda n,s:print(f'adding {n}:{s}'),
>>>                         whenArcAdded=lambda i,j: print(f'adding {i}->{j}'),
>>>                         whenNodeDeleted=lambda n:print(f'deleting {n}'),
>>>                         whenArcDeleted=lambda i,j: print(f'deleting {i}->{j}'))
>>>
>>> ## adding another listener for when a node is deleted
>>> bn.addStructureListener(whenNodeDeleted=lambda n: print('yes, really deleting
↪'+str(n)))
>>>
>>> ## adding nodes to the BN
>>> l=[bn.add(item,3) for item in 'ABCDE']
>>> # adding 0:A
>>> # adding 1:B
>>> # adding 2:C
>>> # adding 3:D
>>> # adding 4:E
>>>
>>> ## adding arc to the BN
>>> bn.addArc(1,3)
>>> # adding 1->3
>>>
>>> ## removing a node from the BN
>>> bn.erase('C')
>>> # deleting 2
>>> # yes, really deleting 2
```

17.1.3 ApproximationSchemeListener

17.1.4 DatabaseGenerationListener

17.2 Random functions

pyAgrum.**initRandom**(*seed=0*)

Initialize random generator seed.

Parameters **seed** (*int*) – the seed used to initialize the random generator

pyAgrum.**randomProba**()

Returns a random number between 0 and 1 included (i.e. a proba).

Return type double

pyAgrum.**randomDistribution**(*n*)

Parameters **n** (*int*) – The number of modalities for the ditribution.

Returns

Return type a random discrete distribution.

17.3 OMP functions

pyAgrum.**isOMP**()

Returns True if OMP has been set at compilation, False otherwise

Return type bool

pyAgrum.**setNumberOfThreads**(*number*)

To avoid spare cycles (less then 100% CPU occupied), use more threads than logical processors (x2 is a good all-around value).

Returns **number** – the number of threads to be used

Return type int

pyAgrum.**getNumberOfLogicalProcessors**()

Returns the number of logical processors

Return type int

pyAgrum.**getMaxNumberOfThreads**()

Returns the max number of threads

Return type int

CHAPTER 18

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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**GumException**(**args*)
Proxy of C++ pyAgrum.Exception class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**FatalError**(**args*)
Proxy of C++ pyAgrum.FatalError class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**FormatNotFound**(**args*)
Proxy of C++ pyAgrum.FormatNotFound class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**InvalidArc**(**args*)
Proxy of C++ pyAgrum.InvalidArc class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**InvalidArgument**(**args*)
Proxy of C++ pyAgrum.InvalidArgument class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**InvalidArgumentsNumber**(**args*)
Proxy of C++ pyAgrum.InvalidArgumentsNumber class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**ArgumentError**(**args*)
Proxy of C++ pyAgrum.ArgumentError class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**SyntaxError**(**args*)
Proxy of C++ pyAgrum.SyntaxError class.

col(*self*)

Returns the indice of the colonne of the error

Return type int

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

line(*self*)

Returns the indice of the line of the error

Return type int

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**UndefinedElement**(**args*)
Proxy of C++ pyAgrum.UndefinedElement class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**UndefinedIteratorKey**(**args*)
Proxy of C++ pyAgrum.UndefinedIteratorKey class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*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(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**DatabaseError**(**args*)
Proxy of C++ pyAgrum.DatabaseError class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyAgrum.**CPTError**(**args*)
Proxy of C++ pyAgrum.CPTError class.

errorCallStack(*self*)

Returns the error call stack

Return type str

errorContent(*self*)

Returns the error content

Return type str

errorType(*self*)

Returns the error type

Return type str

what(*self*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

CHAPTER 19

Configuration for pyAgrum

Configuration for pyAgrum is centralized in an object `gum.config`, singleton of the class `PyAgrumConfiguration`.

class pyAgrum.**PyAgrumConfiguration**

PyAgrumConfiguration is a the pyAgrum configuration singleton. The configuration is build as a classical ConfigParser with read-only structure. Then a value is adressable using a double key: `[section,key]`.

See this notebook (https://lip6.fr/Pierre-Henri.Wuillemin/aGrUM/docs/last/notebooks/configForPyAgrum.ipynb.html).

Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

diff()

print the diff between actual configuration and the defaults. This is what is saved in the file `pyagrum.ini` by the method *PyAgrumConfiguration.save()*

get(*section*, *option*)

Give the value associated to section.option. Preferably use __getitem__ and __setitem__.

Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

Arguments: section {str} – the section option {str} – the property

Returns: str – the value (as string)

grep(*search*)

grep in the configuration any section or properties matching the argument. If a section match the argume, all the section is displayed.

Arguments: search {str} – the string to find

load()
load pyagrum.ini in the current directory, and change the properties if needed

Raises: FileNotFoundError: if there is no pyagrum.ini in the current directory

reset()
back to defaults

save()
Save the diff with the defaults in `pyagrum.ini` in the current directory

set(*section*, *option*, *value*, *no_hook=False*)
set a property in a section. Preferably use `__getitem__` and `__setitem__`.

Examples

```
>>> gum.config['dynamicBN','default_graph_size']=10
>>> gum.config['dynamicBN','default_graph_size']
"10"
```

Arguments: section {str} – the section name (has to exist in defaults) option {str} – the option/property name (has to exist in defaults) value {str} – the value (will be store as string) no_hook {bool} – (optional) should this call trigger the hooks ?

Raises: SyntaxError: if the secion name or the property name does not exist

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X