graphical$_modelsDocumentation$

Chandler Squires

Oct 03, 2022

CONTENTS

CHAPTER

ONE

CLASSES

1.1 AncestralGraph

1.1.1 Overview

class graphical_models.classes.mags.ancestral_graph.**AncestralGraph**(*nodes: Set = frozenset({}), directed: Set = frozenset({}), bidirected: Set = frozenset({}), undirected: Set = frozenset({})*)

Base class for ancestral graphs, used to represent causal models with latent variables.

Copying

AncestralGraph.copy()	Return a copy of this ancestral graph.
AncestralGraph.induced_subgraph(nodes)	Return the induced subgraph over only `nodes`

graphical_models.classes.mags.ancestral_graph.AncestralGraph.copy

AncestralGraph.**copy**()

Return a copy of this ancestral graph.

Returns
A copy of the ancestral graph.

Return type
AncestralGraph

graphical_models.classes.mags.ancestral_graph.AncestralGraph.induced_subgraph

AncestralGraph.**induced_subgraph**(*nodes: Set[Hashable]*)

Return the induced subgraph over only `nodes`

Parameters
`nodes` – Set of nodes for the induced subgraph.

Returns
Induced subgraph over `nodes`.

Return type
AncestralGraph

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(bidirected={(1, 2), (1, 4)}, directed={(1, 3), (2, 3)})
>>> g.induced_subgraph({1, 2, 3})
Directed edges: {(2, 3), (1, 3)}, Bidirected edges: {frozenset({1, 2})}, Undirected␣
↪edges: set()
```

Information about nodes

AncestralGraph.parents_of(nodes)	Return the parents of the node or set of nodes `nodes`.
AncestralGraph.children_of(i)	Return the children of the node or set of nodes `i`.
AncestralGraph.spouses_of(nodes)	Return the spouses of the node or set of nodes `nodes`.
AncestralGraph.neighbors_of(nodes)	Return the neighbors of the node or set of nodes `nodes`.
AncestralGraph.descendants_of(nodes[, ...])	Return the descendants of the node or set of nodes `nodes`.
AncestralGraph.ancestors_of(nodes[, ...])	Return the ancestors of the node or set of nodes `nodes`.
AncestralGraph.district_of(node[, node_subset])	Return the district of a node, i.e., the set of nodes reachable by bidirected edges.
AncestralGraph.markov_blanket_of(node[, flat])	Return the Markov blanket of a node with respect to the whole graph.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.parents_of

AncestralGraph.**parents_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return the parents of the node or set of nodes `nodes`.

Parameters
`nodes` – Nodes.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (2, 3)}, undirected={(1, 4)})
>>> g.parents_of(2)
{1}
>>> g.parents_of({2, 3})
{1, 2}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.children_of

`AncestralGraph.children_of`(*i: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return the children of the node or set of nodes `i`.

Parameters
i – Node.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (2, 3)}, undirected={(1, 4)})
>>> g.children_of(1)
{2}
>>> g.children_of({1, 2})
{2, 3}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.spouses_of

`AncestralGraph.spouses_of`(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return the spouses of the node or set of nodes `nodes`.

Parameters
nodes – Nodes.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (2, 3)}, bidirected={(1, 4), (2, 5)})
>>> g.spouses_of(1)
{4}
>>> g.spouses_of({1, 2})
{4, 5}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.neighbors_of

AncestralGraph.**neighbors_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return the neighbors of the node or set of nodes `nodes`.

Parameters
nodes – Nodes.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 3), (2, 3)}, undirected={(1, 4), (2, 5)})
>>> g.neighbors_of(1)
{4}
>>> g.neighbors_of({1, 2})
{4, 5}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.descendants_of

AncestralGraph.**descendants_of**(*nodes: Union[Hashable, Set[Hashable]], exclude_arcs={}*) → Set[Hashable]

Return the descendants of the node or set of nodes `nodes`.

Parameters
nodes – The nodes.

See also:

ancestors_of

Returns
Return all nodes j such that there is a directed path from node j.

Return type
Set[node]

Example

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.ancestors_of

AncestralGraph.**ancestors_of**(*nodes: Union[Hashable, Set[Hashable]], exclude_arcs={}*) → Set[Hashable]

Return the ancestors of the node or set of nodes `nodes`.

Parameters

- **nodes** – Set of nodes.
- **exclude_arcs** – TODO

See also:

descendants_of

Returns
Return all nodes j such that there is a directed path from j to node.

Return type
Set[node]

Example

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.district_of

`AncestralGraph.`**`district_of`**`(`*node: Hashable*, *node_subset=None*`)` → Set[Hashable]

Return the district of a node, i.e., the set of nodes reachable by bidirected edges. If `node_subset` is provided, do this on the induced subgraph on that subset of nodes.

Returns
The district of node.

Return type
Set[node]

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.markov_blanket_of

`AncestralGraph.`**`markov_blanket_of`**`(`*node*, *flat: bool = False*`)` → Union[Set[Hashable], Dict]

Return the Markov blanket of a node with respect to the whole graph.

Parameters

- **`node`** – The node whose Markov blanket to find.
- **`flat`** – if `True`, return the Markov blanket as a set, otherwise return a dictionary mapping nodes in the district of node to their parents.

Return type
The Markov blanket of node, including the node itself.

Graph modification

AncestralGraph.add_node(node)	Add a node to the ancestral graph.
AncestralGraph.remove_node(node[, ignore_error])	Remove `node`.
AncestralGraph.add_directed(i, j)	Add a directed edge from node `i` to node `j`.
AncestralGraph.remove_directed(i, j[, ...])	Remove the directed edge from `i` to `j`.
AncestralGraph.add_bidirected(i, j)	Add a bidirected edge between nodes `i` and `j`.
AncestralGraph.remove_bidirected(i, j[, ...])	Remove the bidirected edge between `i` and `j`.
AncestralGraph.add_undirected(i, j)	Add an undirected edge between nodes *i* and *j*.
AncestralGraph.remove_undirected(i, j[, ...])	Remove the undirected edge between `i` and `j`.
AncestralGraph.add_nodes_from(nodes)	Add nodes to the ancestral graph.
AncestralGraph.remove_edge(i, j[, ignore_error])	Remove the edge between `i` and `j`, regardless of edge type.
AncestralGraph.remove_edges(edges[, ...])	Remove all edges in `edges` from the graph, regardless of edge type.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.add_node

AncestralGraph.**add_node**(*node: Hashable*)

Add a node to the ancestral graph.

Parameters
node – a hashable Python object

See also:

add_nodes_from

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph()
>>> g.add_node(1)
>>> g.add_node(2)
>>> len(g.nodes)
2
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_node

AncestralGraph.**remove_node**(*node: Hashable*, *ignore_error=False*)

Remove `node`.

Parameters

- **node** – The node to be removed.
- **ignore_error** – If False, raises an error when the node does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(bidirected={(1, 2), (1, 4)}, directed={(1, 3), (2, 3)})
>>> g.remove_node(4)
>>> g
Directed edges: {(2, 3), (1, 3)}, Bidirected edges: {frozenset({1, 2})}, Undirected
↪edges: set()
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.add_directed

AncestralGraph.**add_directed**(*i: Hashable*, *j: Hashable*)

Add a directed edge from node `i` to node `j`.

Parameters

- **i** – source of directed edge.
- **j** – target of directed edge.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph()
>>> g.add_directed(1, 2)
>>> g.directed
{(1, 2)}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_directed

AncestralGraph.**remove_directed**(*i: Hashable*, *j: Hashable*, *ignore_error=False*)

Remove the directed edge from `i` to `j`.

Parameters

- **i** – source of directed edge.
- **j** – target of directed edge.
- **ignore_error** – If False, raises an error when the directed edge does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(bidirected={(1, 2), (1, 4)}, directed={(1, 3), (2, 3)})
>>> g.remove_directed(1, 3)
>>> g
Directed edges: {(2, 3)}, Bidirected edges: {frozenset({1, 4}), frozenset({1, 2})},
↪Undirected edges: set()
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.add_bidirected

`AncestralGraph.add_bidirected`(*i: Hashable*, *j: Hashable*)

Add a bidirected edge between nodes `i` and `j`.

Parameters

- **i** – first endpoint of bidirected edge.
- **j** – second endpoint of bidirected edge.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph()
>>> g.add_bidirected(1, 2)
>>> g.bidirected
{frozenset({i, j})}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_bidirected

`AncestralGraph.remove_bidirected`(*i: Hashable*, *j: Hashable*, *ignore_error=False*)

Remove the bidirected edge between `i` and `j`.

Parameters

- **i** – first endpoint of bidirected edge.
- **j** – second endpoint of bidirected edge.
- **ignore_error** – If False, raises an error when the bidirected edge does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(bidirected={(1, 2), (1, 4)}, directed={(1, 3), (2, 3)})
>>> g.remove_bidirected(1, 2)
>>> g
Directed edges: {(2, 3), (1, 3)}, Bidirected edges: {frozenset({1, 4})}, Undirected␣
↪edges: set()
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.add_undirected

`AncestralGraph.add_undirected`(*i: Hashable*, *j: Hashable*)

Add an undirected edge between nodes *i* and *j*.

Parameters

- **i** – first endpoint of undirected edge.
- **j** – second endpoint of undirected edge.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph()
>>> g.add_undirected(1, 2)
>>> g.undirected
{frozenset({i, j})}
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_undirected

`AncestralGraph.`**`remove_undirected`**(*i: Hashable*, *j: Hashable*, *ignore_error=False*)

Remove the undirected edge between `i` and `j`.

Parameters

- **`i`** – first endpoint of undirected edge.
- **`j`** – second endpoint of undirected edge.
- **`ignore_error`** – If False, raises an error when the undirected edge does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (1, 3)}, undirected={(1, 4)})
>>> g.remove_undirected(1, 4)
>>> g
Directed edges: {(1, 2), (1, 3)}, Bidirected edges: set(), Undirected edges: set()
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.add_nodes_from

`AncestralGraph.`**`add_nodes_from`**(*nodes: Iterable[Hashable]*)

Add nodes to the ancestral graph.

Parameters

`nodes` – an iterable of hashable Python objects

See also:

add_node

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph()
>>> g.add_nodes_from({1, 2})
>>> len(g.nodes)
2
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_edge

`AncestralGraph.remove_edge(i: Hashable, j: Hashable, ignore_error=False)`

Remove the edge between `i` and `j`, regardless of edge type.

Parameters

- **i** – first endpoint of edge.
- **j** – second endpoint of edge.
- **ignore_error** – If False, raises an error when the edge does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (1, 3)}, undirected={(1, 4)})
>>> g.remove_edge(1, 4)
>>> g
Directed edges: {(1, 2), (1, 3)}, Bidirected edges: set(), Undirected edges: set()
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.remove_edges

`AncestralGraph.remove_edges(edges: Iterable, ignore_error=False)`

Remove all edges in `edges` from the graph, regardless of edge type.

Parameters

- **edges** – The edges to be removed from the graph.
- **ignore_error** – If False, raises an error when any edge does not belong to the graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(1, 2), (1, 3)}, undirected={(1, 4)})
>>> g.remove_edges([(1, 4), (1, 2)])
>>> g
Directed edges: {(1, 3)}, Bidirected edges: set(), Undirected edges: set()
```

Graph properties

AncestralGraph.legitimate_mark_changes([...])	Return directed edges that can be changed to bidirected edges, and bidirected edges that can be changed to directed edges.
AncestralGraph.discriminating_triples([verbose])	Return the discriminating triples of the graph, which are triples of nodes that determine the discriminating paths.
AncestralGraph.discriminating_paths([verbose])	TODO
AncestralGraph.is_maximal([new, verbose])	TODO
AncestralGraph.c_components()	Return the c-components of this graph.
AncestralGraph.colliders()	TODO
AncestralGraph.vstructures()	TODO
AncestralGraph.has_directed(i, j)	Check if this graph has the directed edge `i->``j``.
AncestralGraph.has_bidirected(i, j)	Check if this graph has a bidirected edge between `i` and `j`.
AncestralGraph.has_undirected(i, j)	Check if this graph has an undirected edge between `i` and `j`.
AncestralGraph.has_any_edge(i, j)	Check if `i` and `j` are adjacent in this graph.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.legitimate_mark_changes

AncestralGraph.**legitimate_mark_changes**(*verbose=False*, *strict=True*)

Return directed edges that can be changed to bidirected edges, and bidirected edges that can be changed to directed edges.

Parameters

- **verbose** – If True, print each possible mark change and which condition it fails, if any.
- **strict** – If True, check discriminating path condition. Otherwise, check only equality of parents and spouses.

Returns

Directed edges that can be changed to bidirected edges, and bidirected edges that can be changed to directed edges (which will be the new directed edge).

Return type

(mark_changes_dir, mark_changes_bidir)

Example

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(directed={(0, 1)}, bidirected={(1, 2)})
>>> g.legitimate_mark_changes()
({(0, 1)}, {(2, 1)})
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.discriminating_triples

`AncestralGraph.`**`discriminating_triples`**`(`*verbose=False*`)`

Return the discriminating triples of the graph, which are triples of nodes that determine the discriminating paths.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.discriminating_paths

`AncestralGraph.`**`discriminating_paths`**`(`*verbose=False*`)` → Dict[Tuple, str]

TODO

Parameters

`TODO` –

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.is_maximal

`AncestralGraph.`**`is_maximal`**`(`*new=True*, *verbose=False*`)` → bool

TODO

Parameters

`TODO` –

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.c_components

`AncestralGraph.`**`c_components`**`()` → List[set]

Return the c-components of this graph.

Returns

Return the partition of nodes coming from the relation of reachability by bidirected edges.

Return type

List[Set[node]]

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.colliders

`AncestralGraph.colliders()` → set

TODO

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.vstructures

`AncestralGraph.vstructures()` → Set[Tuple]

TODO

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.has_directed

`AncestralGraph.has_directed(`*i: Hashable*, *j: Hashable*`)` → bool

Check if this graph has the directed edge `i->``j``.

See also:

has_bidirected, *has_undirected*, *has_any_edge*

Parameters

- **i** – Node.
- **j** – Node.

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.has_bidirected

`AncestralGraph.has_bidirected(`*i: Hashable*, *j: Hashable*`)` → bool

Check if this graph has a bidirected edge between `i` and `j`.

See also:

has_directed, *has_undirected*, *has_any_edge*

Parameters

- **i** – Node.
- **j** – Node.

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.has_undirected

AncestralGraph.**has_undirected**(*i: Hashable*, *j: Hashable*) → bool

Check if this graph has an undirected edge between `i` and `j`.

See also:

has_directed, *has_bidirected*, *has_any_edge*

Parameters

- **i** – Node.
- **j** – Node.

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.has_any_edge

AncestralGraph.**has_any_edge**(*i: Hashable*, *j: Hashable*) → bool

Check if `i` and `j` are adjacent in this graph.

See also:

has_directed, *has_bidirected*, *has_undirected*

Parameters

- **i** – Node.
- **j** – Node.

Examples

TODO

Ordering

AncestralGraph.topological_sort()	Return a linear order that is consistent with the partial order implied by ancestral relations of this graph.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.topological_sort

AncestralGraph.**topological_sort**() → list

Return a linear order that is consistent with the partial order implied by ancestral relations of this graph.

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(bidirected={(1, 2), (1, 4)}, directed={(1, 3), (2, 3)})
>>> g.topological_sort()
[4, 2, 1, 3]
```

Comparison to other AncestralGraphs

AncestralGraph.shd_skeleton(other)	Compute the structure Hamming distance between the skeleton of this graph and the skeleton of another graph.
AncestralGraph.markov_equivalent(other)	Check if this graph is Markov equivalent to the graph `other`.
AncestralGraph.is_imap(other[, certify])	Check if this graph is an IMAP of the graph `other`, i.e., all m-separation statements in this graph are also m-separation statements in `other`.
AncestralGraph.is_minimal_imap(other[, ...])	TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.shd_skeleton

AncestralGraph.**shd_skeleton**(*other*) → int

Compute the structure Hamming distance between the skeleton of this graph and the skeleton of another graph.

Parameters
other – the graph to which the SHD of the skeleton will be computed.

Returns
The structural Hamming distance between G_1 and G_2 is the minimum number of arc additions, deletions, and reversals required to transform G_1 into G_2 (and vice versa).

Return type
int

Example

```
>>> TODO
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.markov_equivalent

AncestralGraph.**markov_equivalent**(*other*) → bool

Check if this graph is Markov equivalent to the graph `other`. Two graphs are Markov equivalent iff. they have the same skeleton, same v-structures, and if whenever there is the same discriminating path for some node in both graphs, the node is a collider on that path in one graph iff. it is a collider on that path in the other graph.

Parameters
other – another AncestralGraph.

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.is_imap

AncestralGraph.**is_imap**(*other*, *certify: bool = False*) → bool

Check if this graph is an IMAP of the graph `other`, i.e., all m-separation statements in this graph are also m-separation statements in `other`.

Parameters

- **other** – Another DAG.
- **certify** – TODO

See also:

is_minimal_imap

Examples

```
>>> from graphical_models import AncestralGraph
>>> g = AncestralGraph(arcs={(1, 2), (3, 2)})
TODO
```

graphical_models.classes.mags.ancestral_graph.AncestralGraph.is_minimal_imap

AncestralGraph.**is_minimal_imap**(*other*, *certify: bool = False*, *check_imap=True*) → bool

TODO

Parameters
TODO –

Examples

TODO

Separation Statements

AncestralGraph.msep(A, B[, C])	Check whether A and B are m-separated given C, using the Bayes ball algorithm.
AncestralGraph.msep_from_given(A[, C])	Find all nodes m-separated from A given C.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.msep

AncestralGraph.**msep**(*A: Set[Hashable]*, *B: Set[Hashable]*, *C: Set[Hashable] = {}*) → bool

Check whether A and B are m-separated given C, using the Bayes ball algorithm.

Parameters

- **A** – Set
- **B** – Set
- **C** – Set

See also:

msep_from_given

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.msep_from_given

AncestralGraph.**msep_from_given**(*A: Set[Hashable]*, *C: Set[Hashable] = {}*) → Set[Hashable]

Find all nodes m-separated from A given C.

Uses algorithm similar to that in Geiger, D., Verma, T., & Pearl, J. (1990). Identifying independence in Bayesian networks. Networks, 20(5), 507-534.

Parameters

- **A** – Set
- **B** – Set

See also:

msep

Examples

TODO

Conversion to/from other formats

AncestralGraph.to_amat()	Convert the graph into an adjacency matrix.
AncestralGraph.from_amat(amat)	Create a graph from an adjacency matrix.

graphical_models.classes.mags.ancestral_graph.AncestralGraph.to_amat

`AncestralGraph.to_amat()` → numpy.ndarray

Convert the graph into an adjacency matrix. TODO: meaning of numbers

Returns
The adjacency matrix of this graph.

Return type
amat

Examples

TODO

graphical_models.classes.mags.ancestral_graph.AncestralGraph.from_amat

`static AncestralGraph.from_amat`(*amat: numpy.ndarray*)

Create a graph from an adjacency matrix. TODO: meaning of numbers

Parameters
amat – The adjacency matrix

Examples

TODO

1.2 DAG

1.2.1 Copying

DAG.copy()	Return a copy of the current DAG.
DAG.rename_nodes(name_map)	Rename the nodes in this graph according to `name_map`.
DAG.induced_subgraph(nodes)	Return the induced subgraph over only `nodes`

graphical_models.classes.dags.dag.DAG.copy

`DAG.copy()`

Return a copy of the current DAG.

graphical_models.classes.dags.dag.DAG.rename_nodes

`DAG.rename_nodes(name_map: Dict)`

Rename the nodes in this graph according to `name_map`.

Parameters

name_map – A dictionary from the current name of each node to the desired name of each node.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={('a', 'b'), ('b', 'c')})
>>> g2 = g.rename_nodes({'a': 1, 'b': 2, 'c': 3})
>>> g2.arcs
{(1, 2), (2, 3)}
```

graphical_models.classes.dags.dag.DAG.induced_subgraph

`DAG.induced_subgraph(nodes: Set[Hashable])`

Return the induced subgraph over only `nodes`

Parameters

nodes – Set of nodes for the induced subgraph.

Returns

Induced subgraph over `nodes`.

Return type

DAG

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(1, 2), (2, 3), (1, 4)})
>>> d_induced = d.induced_subgraph({1, 2, 3})
>>> d_induced.arcs
{(1, 2), (2, 3)}
```

1.2.2 Information about nodes

DAG.parents_of(nodes)	Return all nodes that are parents of the node or set of nodes `nodes`.
DAG.children_of(nodes)	Return all nodes that are children of the node or set of nodes `nodes`.
DAG.neighbors_of(nodes)	Return all nodes that are adjacent to the node or set of nodes `node`.
DAG.markov_blanket_of(node)	Return the Markov blanket of `node`, i.e., the parents of the node, its children, and the parents of its children.
DAG.ancestors_of(nodes)	Return the ancestors of `nodes`.
DAG.descendants_of(nodes)	Return the descendants of `node`.
DAG.indegree_of(node)	Return the indegree of `node`.
DAG.outdegree_of(node)	Return the outdegree of `node`.
DAG.incoming_arcs(node)	Return all arcs with target `node`.
DAG.outgoing_arcs(node)	Return all arcs with source `node`.
DAG.incident_arcs(node)	Return all arcs with `node` as either source or target.

graphical_models.classes.dags.dag.DAG.parents_of

DAG.**parents_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return all nodes that are parents of the node or set of nodes `nodes`.

Parameters

nodes – A node or set of nodes.

See also:

children_of, *neighbors_of*, *markov_blanket_of*

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 3)})
>>> g.parents_of(2)
{1}
>>> g.parents_of({2, 3})
{1, 2}
```

graphical_models.classes.dags.dag.DAG.children_of

DAG.**children_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return all nodes that are children of the node or set of nodes `nodes`.

Parameters

nodes – A node or set of nodes.

See also:

parents_of, *neighbors_of*, *markov_blanket_of*

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 3)})
>>> g.children_of(1)
{2}
>>> g.children_of({1, 2})
{2, 3}
```

graphical_models.classes.dags.dag.DAG.neighbors_of

DAG.**neighbors_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return all nodes that are adjacent to the node or set of nodes `node`.

Parameters
nodes – A node or set of nodes.

See also:

parents_of, *children_of*, *markov_blanket_of*

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(0,1), (0,2)})
>>> g.neighbors_of(0)
{1, 2}
>>> g.neighbors_of(2)
{0}
```

graphical_models.classes.dags.dag.DAG.markov_blanket_of

DAG.**markov_blanket_of**(*node: Hashable*) → set

Return the Markov blanket of `node`, i.e., the parents of the node, its children, and the parents of its children.

Parameters
node – Node whose Markov blanket to return.

See also:

parents_of, *children_of*, *neighbors_of*

Returns
the Markov blanket of `node`.

Return type
set

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(0, 1), (1, 3), (2, 3), (3, 4)})
>>> g.markov_blanket_of(1)
{0, 2, 3}
```

graphical_models.classes.dags.dag.DAG.ancestors_of

DAG.**ancestors_of**(*nodes: Hashable*) → Set[Hashable]

Return the ancestors of `nodes`.

Parameters
nodes – The node.

See also:

descendants_of

Returns
Return all nodes j such that there is a directed path from j to `node`.

Return type
Set[node]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 3)})
>>> g.ancestors_of(3)
{1, 2, 3}
```

graphical_models.classes.dags.dag.DAG.descendants_of

DAG.**descendants_of**(*nodes: Union[Hashable, Set[Hashable]]*) → Set[Hashable]

Return the descendants of `node`.

Parameters
nodes – The node.

See also:

ancestors_of

Returns
Return all nodes j such that there is a directed path from `node` to j.

Return type
Set[node]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 3)})
>>> g.descendants_of(1)
{2, 3}
```

graphical_models.classes.dags.dag.DAG.indegree_of

`DAG.`**`indegree_of`**(*node: Hashable*) → int

Return the indegree of `node`.

Parameters
node – The node.

See also:

outdegree_of

Returns
The number of parents of `node`.

Return type
int

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.indegree_of(1)
0
>>> g.indegree_of(2)
2
```

graphical_models.classes.dags.dag.DAG.outdegree_of

`DAG.`**`outdegree_of`**(*node: Hashable*) → int

Return the outdegree of `node`.

Parameters
node – The node.

See also:

indegree_of

Returns
The number of children of `node`.

Return type
int

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.outdegree_of(1)
2
>>> g.outdegree_of(3)
0
```

graphical_models.classes.dags.dag.DAG.incoming_arcs

DAG.**incoming_arcs**(*node: Hashable*) → Set[Tuple[Hashable, Hashable]]

Return all arcs with target `node`.

Parameters
`node` – The node.

See also:

incident_arcs, *outgoing_arcs*

Returns
Return all arcs of the form i->``node``.

Return type
Set[arc]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.incoming_arcs(2)
{(1, 2)}
```

graphical_models.classes.dags.dag.DAG.outgoing_arcs

DAG.**outgoing_arcs**(*node: Hashable*) → Set[Tuple[Hashable, Hashable]]

Return all arcs with source `node`.

Parameters
`node` – The node.

See also:

incident_arcs, *incoming_arcs*

Returns
Return all arcs of the form `node`->j.

Return type
Set[arc]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.outgoing_arcs(2)
{(2, 3)}
```

graphical_models.classes.dags.dag.DAG.incident_arcs

DAG.**incident_arcs**(*node: Hashable*) → Set[Tuple[Hashable, Hashable]]

Return all arcs with `node` as either source or target.

Parameters
node – The node.

See also:

incoming_arcs, *outgoing_arcs*

Returns
Return all arcs i->j such that either i=``node`` of j=``node``.

Return type
Set[arc]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.incident_arcs(2)
{(1, 2), (2, 3)}
```

1.2.3 Graph modification

DAG.add_node(node)	Add `node` to the DAG.
DAG.add_nodes_from(nodes)	Add nodes to the graph from the collection `nodes`.
DAG.remove_node(node[, ignore_error])	Remove the node `node` from the graph.
DAG.add_arc(i, j[, check_acyclic])	Add the arc `i -> j` to the DAG
DAG.add_arcs_from(arcs[, check_acyclic])	Add arcs to the graph from the collection `arcs`.
DAG.remove_arc(i, j[, ignore_error])	Remove the arc `i -> j`.
DAG.reverse_arc(i, j[, ignore_error, ...])	Reverse the arc `i -> j` to `i <- j`.

graphical_models.classes.dags.dag.DAG.add_node

`DAG.`**`add_node`**(*node: Hashable*)

Add node to the DAG.

Parameters
`node` – a hashable Python object

See also:

`add_nodes_from`

Examples

```
>>> from graphical_models import DAG
>>> g = DAG()
>>> g.add_node(1)
>>> g.add_node(2)
>>> len(g.nodes)
2
```

graphical_models.classes.dags.dag.DAG.add_nodes_from

`DAG.`**`add_nodes_from`**(*nodes: Iterable*)

Add nodes to the graph from the collection `nodes`.

Parameters
`nodes` – collection of nodes to be added.

See also:

`add_node`

Examples

```
>>> from graphical_models import DAG
>>> g = DAG({1, 2})
>>> g.add_nodes_from({'a', 'b'})
>>> g.add_nodes_from(range(3, 6))
>>> g.nodes
{1, 2, 'a', 'b', 3, 4, 5}
```

graphical_models.classes.dags.dag.DAG.remove_node

`DAG.`**`remove_node`**(*node: Hashable*, *ignore_error=False*)

Remove the node `node` from the graph.

Parameters

- **`node`** – node to be removed.
- **`ignore_error`** – if True, ignore the KeyError raised when node is not in the DAG.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2)})
>>> g.remove_node(2)
>>> g.nodes
{1}
```

graphical_models.classes.dags.dag.DAG.add_arc

DAG.**add_arc**(*i: Hashable*, *j: Hashable*, *check_acyclic=True*)

Add the arc `i -> j` to the DAG

Parameters

- **i** – source node of the arc
- **j** – target node of the arc
- **check_acyclic** – if True, check that the DAG remains acyclic after adding the edge.

See also:

add_arcs_from

Examples

```
>>> from graphical_models import DAG
>>> g = DAG({1, 2})
>>> g.add_arc(1, 2)
>>> g.arcs
{(1, 2)}
```

graphical_models.classes.dags.dag.DAG.add_arcs_from

DAG.**add_arcs_from**(*arcs: Iterable[Tuple]*, *check_acyclic=False*)

Add arcs to the graph from the collection `arcs`.

Parameters

- **arcs** – collection of arcs to be added.
- **check_acyclic** – if True, check that the DAG remains acyclic after adding the edge.

See also:

`add_arcs`

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2)})
>>> g.add_arcs_from({(1, 3), (2, 3)})
>>> g.arcs
{(1, 2), (1, 3), (2, 3)}
```

graphical_models.classes.dags.dag.DAG.remove_arc

DAG.**remove_arc**(*i: Hashable*, *j: Hashable*, *ignore_error=False*)

Remove the arc `i -> j`.

Parameters

- **i** – source of arc to be removed.
- **j** – target of arc to be removed.
- **ignore_error** – if True, ignore the KeyError raised when arc is not in the DAG.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2)})
>>> g.remove_arc(1, 2)
>>> g.arcs
set()
```

graphical_models.classes.dags.dag.DAG.reverse_arc

DAG.**reverse_arc**(*i: Hashable*, *j: Hashable*, *ignore_error=False*, *check_acyclic=False*)

Reverse the arc `i -> j` to `i <- j`.

Parameters

- **i** – source of arc to be reversed.
- **j** – target of arc to be reversed.
- **ignore_error** – if True, ignore the KeyError raised when arc is not in the DAG.
- **check_acyclic** – if True, check that the DAG remains acyclic after adding the edge.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2)})
>>> g.reverse_arc(1, 2)
>>> g.arcs
{(2, 1)}
```

1.2.4 Graph properties

DAG.has_arc(source, target)	Check if this DAG has an arc `source -> target`.
DAG.sources()	Get all nodes in the graph that have no parents.
DAG.sinks()	Get all nodes in the graph that have no children.
DAG.reversible_arcs()	Get all reversible (aka covered) arcs in the DAG.
DAG.is_reversible(i, j)	Check if the arc `i -> j` is reversible (aka covered), i.e., if $pa(i) = pa(j) \setminus \{i\}$
DAG.arcs_in_vstructures()	Get all arcs in the graph that participate in a v-structure.
DAG.vstructures()	Get all v-structures in the graph, i.e., triples of the form (i, k, j) such that `i->k<-j` and `i` is not adjacent to `j`.
DAG.triples()	Return all triples of the form (`i`, `j`, `k`) such that `i` and `k` are both adjacent to `j`.
DAG.upstream_most(s)	Return the set of nodes which in `s` which have no ancestors in `s`.

graphical_models.classes.dags.dag.DAG.has_arc

DAG.**has_arc**(*source: Hashable*, *target: Hashable*) → bool

Check if this DAG has an arc `source -> target`.

Parameters

- **source** – Source node of arc.
- **target** – Target node of arc.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(0,1), (0,2)})
>>> g.has_arc(0, 1)
True
>>> g.has_arc(1, 2)
False
```

graphical_models.classes.dags.dag.DAG.sources

DAG.**sources**() → Set[Hashable]

Get all nodes in the graph that have no parents.

Returns
Nodes in the graph that have no parents.

Return type
List[node]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.sources()
{1}
```

graphical_models.classes.dags.dag.DAG.sinks

`DAG.sinks()` → Set[Hashable]

Get all nodes in the graph that have no children.

Returns
Nodes in the graph that have no children.

Return type
List[node]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.sinks()
{3}
```

graphical_models.classes.dags.dag.DAG.reversible_arcs

`DAG.reversible_arcs()` → Set[Tuple[Hashable, Hashable]]

Get all reversible (aka covered) arcs in the DAG.

Returns
Return all reversible (aka covered) arcs in the DAG. An arc i -> j is *covered* if the $Pa(j) = Pa(i) \cup i$. Reversing a reversible arc results in a DAG in the same Markov equivalence class.

Return type
Set[arc]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.reversible_arcs()
{(1, 2), (2, 3)}
```

graphical_models.classes.dags.dag.DAG.is_reversible

`DAG.is_reversible(i: Hashable, j: Hashable) → bool`

Check if the arc `i -> j` is reversible (aka covered), i.e., if $pa(i) = pa(j) \setminus \{i\}$

Parameters

- **i** – source of the arc
- **j** – target of the arc

Return type

True if the arc is reversible, otherwise False.

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.is_reversible(1, 2)
True
>>> g.is_reversible(1, 3)
False
```

graphical_models.classes.dags.dag.DAG.arcs_in_vstructures

`DAG.arcs_in_vstructures() → Set[Tuple]`

Get all arcs in the graph that participate in a v-structure.

Returns

Return all arcs in the graph in a v-structure (aka an immorality). A v-structure is formed when i->j<-k but there is no arc between i and k. Arcs that participate in a v-structure are identifiable from observational data.

Return type

Set[arc]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 3), (2, 3)})
>>> g.arcs_in_vstructures()
{(1, 3), (2, 3))
```

graphical_models.classes.dags.dag.DAG.vstructures

DAG.**vstructures**() → Set[Tuple]

Get all v-structures in the graph, i.e., triples of the form (i, k, j) such that `i->k<-j` and `i` is not adjacent to `j`.

Returns
Return all triples in the graph in a v-structure (aka an immorality). A v-structure is formed when i->j<-k but there is no arc between i and k. Arcs that participate in a v-structure are identifiable from observational data.

Return type
Set[Tuple]

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 3), (2, 3)})
>>> g.vstructures()
{(1, 3, 2)}
```

graphical_models.classes.dags.dag.DAG.triples

DAG.**triples**() → Set[Tuple]

Return all triples of the form (`i`, `j`, `k`) such that `i` and `k` are both adjacent to `j`.

Returns
Triples in the graph.

Return type
Set[Tuple]

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 3), (2, 3), (1, 2)})
>>> g.triples()
{frozenset({1, 3, 2})}
```

graphical_models.classes.dags.dag.DAG.upstream_most

DAG.**upstream_most**(*s: Set[Hashable]*) → Set[Hashable]

Return the set of nodes which in `s` which have no ancestors in `s`.

Parameters
s – Set of nodes

Return type
The set of nodes in `s` with no ancestors in `s`.

1.2.5 Ordering

DAG.topological_sort()	Return a topological sort of the nodes in the graph.
DAG.is_topological(order)	Check that `order` is a topological order consistent with this DAG, i.e., if `i->``j``` in the DAG, then `i` comes before `j` in the order.
DAG.permutation_score(order)	Return the number of "errors" in `order` with respect to the DAG, i.e., the number of times that `i->``j``` in the DAG but `i` comes *after* `j` in `order`.

graphical_models.classes.dags.dag.DAG.topological_sort

DAG.**topological_sort**() → List[Hashable]

Return a topological sort of the nodes in the graph.

Returns
A topological sort of the nodes in a graph.

Return type
List[Node]

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 3)})
>>> g.topological_sort
[1, 2, 3]
```

graphical_models.classes.dags.dag.DAG.is_topological

DAG.**is_topological**(*order: list*) → bool

Check that `order` is a topological order consistent with this DAG, i.e., if `i->``j``` in the DAG, then `i` comes before `j` in the order.

Parameters
order – the order to check.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3)})
>>> g.is_topological([1, 2, 3])
True
>>> g.is_topological([1, 3, 2])
True
>>> g.is_topological([2, 1, 3])
False
```

graphical_models.classes.dags.dag.DAG.permutation_score

DAG.**permutation_score**(*order: list*) → int

Return the number of "errors" in `order` with respect to the DAG, i.e., the number of times that `i->``j`` in the DAG but `i` comes *after* `j` in `order`.

Parameters
order – the order to check.

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3)})
>>> g.permutation_score([1, 2, 3])
0
>>> g.permutation_score([2, 1, 3])
1
>>> g.permutation_score([2, 3, 1])
2
```

1.2.6 Comparison to other DAGs

DAG.shd(other)	Compute the structural Hamming distance between this DAG and the DAG `other`.
DAG.shd_skeleton(other)	Compute the structure Hamming distance between the skeleton of this DAG and the skeleton of the graph `other`.
DAG.markov_equivalent(other[, interventions])	Check if this DAG is (interventionally) Markov equivalent to the DAG `other`.
DAG.is_imap(other)	Check if this DAG is an IMAP of the DAG `other`, i.e., all d-separation statements in this graph are also d-separation statements in `other`.
DAG.is_minimal_imap(other[, certify, check_imap])	Check if this DAG is a minimal IMAP of *other*, i.e., it is an IMAP and no proper subgraph of this DAG is an IMAP of other.
DAG.chickering_distance(other)	Return the total number of edge reversals plus twice the number of edge additions/deletions required to turn this DAG into the DAG `other`.
DAG.confusion_matrix(other[, rates_only])	Return the "confusion matrix" associated with estimating the CPDAG of `other` instead of the CPDAG of this DAG.
DAG.confusion_matrix_skeleton(other)	Return the "confusion matrix" associated with estimating the skeleton of `other` instead of the skeleton of this DAG.

graphical_models.classes.dags.dag.DAG.shd

DAG.**shd**(*other*) → int

Compute the structural Hamming distance between this DAG and the DAG `other`.

Parameters
other – the DAG to which the SHD will be computed.

Returns
The structural Hamming distance between G_1 and G_2 is the minimum number of arc additions, deletions, and reversals required to transform G_1 into G_2 (and vice versa).

Return type
int

Example

```
>>> from graphical_models import DAG
>>> g1 = DAG(arcs={(1, 2), (2, 3)})
>>> g2 = DAG(arcs={(2, 1), (2, 3)})
>>> g1.shd(g2)
1
```

graphical_models.classes.dags.dag.DAG.shd_skeleton

DAG.**shd_skeleton**(*other*) → int

Compute the structure Hamming distance between the skeleton of this DAG and the skeleton of the graph `other`.

Parameters
other – the DAG to which the SHD of the skeleton will be computed.

Returns
The structural Hamming distance between G_1 and G_2 is the minimum number of arc additions, deletions, and reversals required to transform G_1 into G_2 (and vice versa).

Return type
int

Example

```
>>> from graphical_models import DAG
>>> g1 = DAG(arcs={(1, 2), (2, 3)})
>>> g2 = DAG(arcs={(2, 1), (2, 3)})
>>> g1.shd_skeleton(g2)
0
```

```
>>> g1 = DAG(arcs={(1, 2)})
>>> g2 = DAG(arcs={(1, 2), (2, 3)})
>>> g1.shd_skeleton(g2)
1
```

graphical_models.classes.dags.dag.DAG.markov_equivalent

DAG.**markov_equivalent**(*other*, *interventions=None*) → bool

Check if this DAG is (interventionally) Markov equivalent to the DAG `other`.

Parameters

- **other** – Another DAG.
- **interventions** – If not None, check whether the two DAGs are interventionally Markov equivalent under the interventions.

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(0, 1), (1, 2)})
>>> d2 = DAG(arcs={(2, 1), (1, 0)})
>>> d3 = DAG(arcs={(0, 1), (2, 1)})
>>> d4 = DAG(arcs={(1, 0), (1, 2)})
>>> d1.markov_equivalent(d2)
True
>>> d2.markov_equivalent(d1)
True
>>> d1.markov_equivalent(d3)
False
>>> d1.markov_equivalent(d2, [{2}])
False
>>> d1.markov_equivalent(d4, [{2}])
True
```

graphical_models.classes.dags.dag.DAG.is_imap

DAG.**is_imap**(*other*) → bool

Check if this DAG is an IMAP of the DAG `other`, i.e., all d-separation statements in this graph are also d-separation statements in `other`.

Parameters
other – Another DAG.

See also:

is_minimal_imap

Returns
True if `other` is an I-MAP of this DAG, otherwise False.

Return type
bool

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (3, 2)})
>>> other = DAG(arcs={(1, 2)})
>>> g.is_imap(other)
True
>>> other = DAG(arcs={(1, 2), (2, 3)})
>>> g.is_imap(other)
False
```

graphical_models.classes.dags.dag.DAG.is_minimal_imap

DAG.**is_minimal_imap**(*other*, *certify=False*, *check_imap=True*) → Union[bool, Tuple[bool, Any]]

Check if this DAG is a minimal IMAP of *other*, i.e., it is an IMAP and no proper subgraph of this DAG is an IMAP of other. Deleting the arc i->j retains IMAPness when *i* is d-separated from *j* in *other* given the parents of *j* besides *i* in this DAG.

Parameters

- **other** – Another DAG.
- **certify** – If True and this DAG is not an IMAP of other, return a certificate of non-minimality in the form of an edge i->j that can be deleted while retaining IMAPness.
- **check_imap** – If True, first check whether this DAG is an IMAP of other, if False, this DAG is assumed to be an IMAP of other.

See also:

is_imap

Returns
True if `other` is a minimal I-MAP of this DAG, otherwise False.

Return type
bool

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (3, 2)})
>>> other = DAG(arcs={(1, 2)})
>>> g.is_minimal_imap(other)
False
```

graphical_models.classes.dags.dag.DAG.chickering_distance

`DAG.chickering_distance(other)` → int

Return the total number of edge reversals plus twice the number of edge additions/deletions required to turn this DAG into the DAG `other`.

Parameters
`other` – the DAG against which to compare the Chickering distance.

Returns
The Chickering distance between this DAG and the DAG `other`.

Return type
int

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(0, 1), (1, 2)})
>>> d2 = DAG(arcs={(0, 1), (2, 1), (3, 1)})
>>> d1.chickering_distance(d2)
3
```

graphical_models.classes.dags.dag.DAG.confusion_matrix

`DAG.confusion_matrix(other, rates_only=False)`

Return the "confusion matrix" associated with estimating the CPDAG of `other` instead of the CPDAG of this DAG.

Parameters

- `other` – The DAG against which to compare.
- `rates_only` – if True, the dictionary of results only contains the false positive rate, true positive rate, and precision.

Returns

Dictionary of results

- **false_positive_arcs:**
  the arcs in the CPDAG of `other` which are not arcs or edges in the CPDAG of this DAG.
- **false_positive_edges:**
  the edges in the CPDAG of `other` which are not arcs or edges in the CPDAG of this DAG.
- **false_negative_arcs:**
  the arcs in the CPDAG of this graph which are not arcs or edges in the CPDAG of `other`.
- **true_positive_arcs:**
  the arcs in the CPDAG of `other` which are arcs in the CPDAG of this DAG.
- **reversed_arcs:**
  the arcs in the CPDAG of `other` whose reversals are arcs in the CPDAG of this DAG.
- **mistaken_arcs_for_edges:**
  the arcs in the CPDAG of `other` whose reversals are arcs in the CPDAG of this DAG.

- **false_negative_edges:**
  the edges in the CPDAG of this DAG which are not arcs or edges in the CPDAG of `other`.
- **true_positive_edges:**
  the edges in the CPDAG of `other` which are edges in the CPDAG of this DAG.
- **mistaken_edges_for_arcs:**
  the edges in the CPDAG of `other` which are arcs in the CPDAG of this DAG.
- **num_false_positives:**
  the total number of: false_positive_arcs, false_positive_edges
- **num_false_negatives:**
  the total number of: false_negative_arcs, false_negative_edges, mistaken_arcs_for_edges, and reversed_arcs
- **num_true_positives:**
  the total number of: true_positive_arcs, true_positive_edges, and mistaken_edges_for_arcs
- **num_true_negatives:**
  the total number of missing arcs/edges in `other` which are actually missing in this DAG.
- **fpr:**
  the false positive rate, i.e., num_false_positives/(num_false_positives+num_true_negatives). If this DAG is fully connected, defaults to 0.
- **tpr:**
  the true positive rate, i.e., num_true_positives/(num_true_positives+num_false_negatives). If this DAG is empty, defaults to 1.
- **precision:**
  the precision, i.e., num_true_positives/(num_true_positives+num_false_positives). If `other` is empty, defaults to 1.

Return type
dict

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(0, 1), (1, 2)})
>>> d2 = DAG(arcs={(0, 1), (2, 1)})
>>> cm = d1.confusion_matrix(d2)
>>> cm["mistaken_edges_for_arcs"]
{frozenset({0, 1}), frozenset({1, 2})},
>>> cm = d2.confusion_matrix(d1)
>>> cm["mistaken_arcs_for_edges"]
{(0, 1), (2, 1)}
```

graphical_models.classes.dags.dag.DAG.confusion_matrix_skeleton

DAG.**confusion_matrix_skeleton**(*other*)

Return the "confusion matrix" associated with estimating the skeleton of `other` instead of the skeleton of this DAG.

Parameters
`other` – The DAG against which to compare.

Returns

Dictionary of results

- **false_positives:**
  the edges in the skeleton of `other` which are not in the skeleton of this DAG.
- **false_negatives:**
  the edges in the skeleton of this graph which are not in the skeleton of `other`.
- **true_positives:**
  the edges in the skeleton of `other` which are acutally in the skeleton of this DAG.
- **num_false_positives:**
  the total number of false_positives
- **num_false_negatives:**
  the total number of false_negatives
- **num_true_positives:**
  the total number of true_positives
- **num_true_negatives:**
  the total number of missing edges in the skeleton of `other` which are actually missing in this DAG.
- **fpr:**
  the false positive rate, i.e., num_false_positives/(num_false_positives+num_true_negatives). If this DAG is fully connected, defaults to 0.
- **tpr:**
  the true positive rate, i.e., num_true_positives/(num_true_positives+num_false_negatives). If this DAG is empty, defaults to 1.
- **precision:**
  the precision, i.e., num_true_positives/(num_true_positives+num_false_positives). If `other` is empty, defaults to 1.

Return type
dict

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(0, 1), (1, 2)})
>>> d2 = DAG(arcs={(0, 1), (2, 1)})
>>> cm = d1.confusion_matrix_skeleton(d2)
>>> cm["tpr"]
1.0
>>> d3 = DAG(arcs={(0, 1), (0, 2)})
```

(continues on next page)

(continued from previous page)

```
>>> cm = d2.confusion_matrix_skeleton(d3)
>>> cm["true_positives"]
{frozenset({0, 1})}
>>> cm["false_positives"]
{frozenset({0, 2})},
>>> cm["false_negatives"]
{frozenset({1, 2})}
```

1.2.7 Separation statements

DAG.dsep(A, B[, C, verbose, certify])	Check if A and B are d-separated given C, using the Bayes ball algorithm.
DAG.dsep_from_given(A[, C])	Find all nodes d-separated from A given C.
DAG.is_invariant(A, intervened_nodes[, ...])	Check if the distribution of A given cond_set is invariant to an intervention on intervened_nodes.
DAG.local_markov_statements()	Return the local Markov statements of this DAG, i.e., those of the form i independent nondescendants(i) given the parents of i.

graphical_models.classes.dags.dag.DAG.dsep

DAG.**dsep**(*A: Union[Set[Hashable], Hashable]*, *B: Union[Set[Hashable], Hashable]*, *C: Union[Set[Hashable], Hashable] = {}*, *verbose=False*, *certify=False*) → bool

Check if A and B are d-separated given C, using the Bayes ball algorithm.

Parameters

- **A** – First set of nodes.
- **B** – Second set of nodes.
- **C** – Separating set of nodes.
- **verbose** – If True, print moves of the algorithm.

See also:

dsep_from_given

Return type

is_dsep

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (3, 2)})
>>> g.dsep(1, 3)
True
>>> g.dsep(1, 3, 2)
False
```

graphical_models.classes.dags.dag.DAG.dsep_from_given

DAG.**dsep_from_given**(*A*, *C: Union[Hashable, Set[Hashable]] = frozenset({})*) → Set[Hashable]

Find all nodes d-separated from `A` given `C`.

Uses algorithm in Geiger, D., Verma, T., & Pearl, J. (1990). Identifying independence in Bayesian networks. Networks, 20(5), 507-534.

Parameters

- **A** – set of nodes.
- **C** – set of conditioned nodes.

Returns
Nodes which are d-separated from `A` given `C`.

Return type
set

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (2, 3), (3, 4)})
>>> d.dsep_from_given(0, 1)
{2, 3, 4]
```

graphical_models.classes.dags.dag.DAG.is_invariant

DAG.**is_invariant**(*A*, *intervened_nodes*, *cond_set={}*, *verbose=False*) → bool

Check if the distribution of `A` given cond_set is invariant to an intervention on intervened_nodes.

$f^\emptyset(A|C) = f^I(A|C)$ if the "intervention node" I with intervened_nodes as its children is d-separated from A given C. Equivalently, the :math:`f^emptyset(A|C)

eq f^I(A|C)` if:

- **there is an active path to an intervened node that ends in an arrowhead, and that intervened node**
  or one of its descendants is conditioned on.
- **there is an active path to an intervened node that ends in a tail, and that intervened node**
  is not conditioned on.

A:
Set of nodes.

intervened_nodes:
Nodes on which an intervention has occurred.

cond_set:
Conditioning set for the tested distribution.

verbose:
If True, print moves of the algorithm.

graphical_models.classes.dags.dag.DAG.local_markov_statements

`DAG.local_markov_statements()` → Set[Tuple[Any, FrozenSet, FrozenSet]]

Return the local Markov statements of this DAG, i.e., those of the form `i` independent nondescendants(i) given the parents of `i`.

Returns
The set of tuples of the form (`i`, `A`, `C`) representing the local Markov statements of the DAG via (`i` independent of `A` given `C`).

Return type
set

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (3, 2)})
>>> g.local_markov_statements()
{(1, frozenset({3}), frozenset()), (2, frozenset(), frozenset({1, 3})), (3,␣
↪frozenset({1}), frozenset())}
```

1.2.8 Conversion to/from other formats

DAG.from_amat(amat)	Return a DAG with arcs given by `amat`, i.e. i->j if `amat[i,j] != 0`.
DAG.to_amat([node_list])	Return an adjacency matrix for this DAG.
DAG.from_nx(nx_graph)	Convert a networkx DiGraph into a DAG.
DAG.to_nx()	Convert DAG to a networkx DiGraph.
DAG.from_dataframe(df)	Create a DAG from a dataframe, where the indices and columns are node names and a nonzero entry indicates the presence of an edge.
DAG.to_dataframe([node_list])	Turn this DAG into a dataframe, where the indices and columns are node names and a nonzero entry indicates the presence of an edge.

graphical_models.classes.dags.dag.DAG.from_amat

classmethod `DAG.from_amat`(*amat: numpy.ndarray*)

Return a DAG with arcs given by `amat`, i.e. i->j if `amat[i,j] != 0`.

Parameters
amat – Numpy matrix representing arcs in the DAG.

Examples

```
>>> from graphical_models import DAG
>>> import numpy as np
>>> amat = np.array([[0, 0, 1], [0, 0, 1], [0, 0, 0]])
>>> d = DAG.from_amat(amat)
>>> d.arcs
{(0, 2), (1, 2)}
```

graphical_models.classes.dags.dag.DAG.to_amat

DAG.**to_amat**(*node_list=None*) -> *(numpy.ndarray*, *<class 'list'>)*

Return an adjacency matrix for this DAG.

Parameters
node_list – List indexing the rows/columns of the matrix.

See also:

from_amat

Return type
(amat, node_list)

Example

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.to_amat()[0]
array([[0, 1, 1],
       [0, 0, 1],
       [0, 0, 0]])
>>> g.to_amat()[1]
[1, 2, 3]
```

graphical_models.classes.dags.dag.DAG.from_nx

classmethod DAG.**from_nx**(*nx_graph: networkx.DiGraph*)

Convert a networkx DiGraph into a DAG.

Parameters
nx_graph – networkx DiGraph

Returns
The graph as a DAG object.

Return type
DAG

Examples

```
>>> from graphical_models import DAG
>>> import networkx as nx
>>> g = nx.DiGraph()
>>> g.add_edges_from([(0, 1)])
>>> d = DAG.from_nx(g)
>>> d.arcs
{(0, 1)}
```

graphical_models.classes.dags.dag.DAG.to_nx

DAG.**to_nx**() → networkx.DiGraph

Convert DAG to a networkx DiGraph.

Returns
The graph as a networkx.DiGraph object.

Return type
networkx.DiGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1)})
>>> g = d.to_nx()
>>> g.edges
OutEdgeView([(0, 1)])
```

graphical_models.classes.dags.dag.DAG.from_dataframe

classmethod DAG.**from_dataframe**(*df*)

Create a DAG from a dataframe, where the indices and columns are node names and a nonzero entry indicates the presence of an edge.

Parameters
df – The pandas dataframe.

Returns
The graph as a DAG object.

Return type
DAG

Examples

```
>>> from graphical_models import DAG
>>> import numpy as np
>>> import pandas as pd
>>> amat = np.array([[0, 1], [0, 0]])
>>> df = pd.DataFrame(amat, index=["a", "b"], columns=["a", "b"])
>>> d = DAG.from_dataframe(df)
>>> d.arcs
{('a', 'b')}
```

graphical_models.classes.dags.dag.DAG.to_dataframe

DAG.**to_dataframe**(*node_list=None*)

Turn this DAG into a dataframe, where the indices and columns are node names and a nonzero entry indicates the presence of an edge.

Parameters
node_list – Order to use when creating the dataframe. If None, uses a sorted order.

Returns
The graph as a DataFrame.

Return type
pandas.DataFrame

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1)})
>>> d.to_dataframe()
   0  1
0  0  1
1  0  0
>>> d.to_dataframe(node_list=[1, 0])
   1  0
1  0  0
0  1  0
```

1.2.9 Conversion to other graphs

DAG.moral_graph()	Return the (undirected) moral graph of this DAG, i.e., the graph with the parents of all nodes made adjacent.
DAG.marginal_mag(latent_nodes[, relabel, new])	Return the maximal ancestral graph (MAG) that results from marginalizing out `latent_nodes`.
DAG.cpdag()	Return the completed partially directed acyclic graph (CPDAG, aka essential graph) that represents the Markov equivalence class of this DAG.
DAG.interventional_cpdag(interventions[, cpdag])	Return the interventional essential graph (aka CPDAG) associated with this DAG.

graphical_models.classes.dags.dag.DAG.moral_graph

DAG.**moral_graph**()

Return the (undirected) moral graph of this DAG, i.e., the graph with the parents of all nodes made adjacent.

Returns
Moral graph of this DAG.

Return type
UndirectedGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(1, 3), (2, 3)})
>>> ug = d.moral_graph()
>>> ug.edges
{frozenset({1, 3}), frozenset({2, 3}), frozenset({1, 2})}
```

graphical_models.classes.dags.dag.DAG.marginal_mag

DAG.**marginal_mag**(*latent_nodes*, *relabel=None*, *new=True*)

Return the maximal ancestral graph (MAG) that results from marginalizing out `latent_nodes`.

Parameters

- **latent_nodes** – nodes to marginalize over.
- **relabel** – if relabel='default', relabel the nodes to have labels 1,2,. . . ,(#nodes).
- **new** – TODO - pick whether to use new or old implementation.

Returns
AncestralGraph, the MAG resulting from marginalizing out *latent_nodes*.

Return type
m

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(1, 3), (1, 2)})
>>> mag = d.marginal_mag(latent_nodes={1})
>>> mag
Directed edges: set(), Bidirected edges: {frozenset({2, 3})}, Undirected edges:␣
↪set()
>>> mag = d.marginal_mag(latent_nodes={1}, relabel="default")
Directed edges: set(), Bidirected edges: {frozenset({0, 1})}, Undirected edges:␣
↪set()
```

graphical_models.classes.dags.dag.DAG.cpdag

DAG.**cpdag**()

Return the completed partially directed acyclic graph (CPDAG, aka essential graph) that represents the Markov equivalence class of this DAG.

Returns
CPDAG representing the MEC of this DAG.

Return type
causaldag.PDAG

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 4), (3, 4)})
>>> cpdag = g.cpdag()
>>> cpdag.edges
{frozenset({1, 2})}
>>> cpdag.arcs
{(2, 4), (3, 4)}
```

graphical_models.classes.dags.dag.DAG.interventional_cpdag

DAG.**interventional_cpdag**(*interventions: List[set]*, *cpdag=None*)

Return the interventional essential graph (aka CPDAG) associated with this DAG.

Parameters

- **interventions** – A list of the intervention targets.
- **cpdag** – The original (non-interventional) CPDAG of the graph. Faster when provided.

Returns
Interventional CPDAG representing the I-MEC of this DAG.

Return type
causaldag.PDAG

Examples

```
>>> from graphical_models import DAG
>>> g = DAG(arcs={(1, 2), (2, 4), (3, 4)})
>>> cpdag = g.cpdag()
>>> icpdag = g.interventional_cpdag([{1}], cpdag=cpdag)
>>> icpdag.arcs
{(1, 2), (2, 4), (3, 4)}
```

1.2.10 Chickering Sequences

DAG.resolved_sinks(other)	Return the nodes in this graph which are "resolved sinks" with respect to the graph `other`.
DAG.chickering_sequence(imap[, verbose])	Return a *Chickering sequence* from this DAG to an I-MAP `imap`.
DAG.apply_edge_operation(imap[, seed_sink, ...])	Identify an edge operation (covered edge reversal or edge addition) which decreases the Chickering distance from this DAG to `imap`.

graphical_models.classes.dags.dag.DAG.resolved_sinks

DAG.**resolved_sinks**(*other*) → set

Return the nodes in this graph which are "resolved sinks" with respect to the graph `other`.

A "resolved sink" is a node which has the same parents in both graphs, and no children which are not themselves resolved sinks.

Parameters
other – TODO

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(1, 0), (1, 2), (2, 0)})
>>> d2 = DAG(arcs={(2, 0), (2, 1), (1, 0)})
>>> res_sinks = d1.resolved_sinks(d2)
{0}
```

graphical_models.classes.dags.dag.DAG.chickering_sequence

DAG.**chickering_sequence**(*imap*, *verbose=False*)

Return a *Chickering sequence* from this DAG to an I-MAP `imap`.

A Chickering sequence from DAG `D1` to a DAG `D2` is a sequence of DAGs starting at `D1` and ending at `D2`, with consecutive DAGs differing by a single edge reversal or edge deletion, such that each DAG is an IMAP of `D1`.

See Chickering, David Maxwell. "Optimal structure identification with greedy search." (2002) for more details.

Parameters
imap (*DAG*) – The I-MAP of this DAG at which the Chickering sequence will end.

Examples

```
>>> from graphical_models import DAG
>>> d1 = DAG(arcs={(0, 1), (1, 2)})
>>> d2 = DAG(arcs={(2, 0), (2, 1), (1, 0)})
>>> sequence, moves = d1.chickering_sequence(d2)
>>> sequence[1].arcs
{(1, 0), (1, 2)}
>>> sequence[2].arcs
{(1, 0), (1, 2), (2, 0)}
>>> moves
[
    {'sink': 0, 'move': 6, 'd': 2},
    {'sink': 0, 'move': 4},
    {'sink': 1, 'move': 6, 'd': 2}
]
```

graphical_models.classes.dags.dag.DAG.apply_edge_operation

DAG.**apply_edge_operation**(*imap*, *seed_sink=None*, *verbose=False*)

Identify an edge operation (covered edge reversal or edge addition) which decreases the Chickering distance from this DAG to imap.

See Chickering, David Maxwell. "Optimal structure identification with greedy search." (2002), Fig. 2 for more details.

Parameters

- **imap** – The target I-MAP.
- **seed_sink** – If the algorithm reaches step 3, pick this node (if it is indeed a valid sink).
- **verbose** – If True, print out the steps of the algorithm.

Returns

- The updated DAG
- The node picked for the operation
- The type of the edge operation (corresponding to the line of the algorithm in the above paper)

Return type

(DAG, Node, int)

1.2.11 Directed Clique Trees

DAG.directed_clique_tree([verbose])	Return the directed clique tree associated with this DAG.
DAG.contracted_directed_clique_tree()	Return the contracted directed clique tree associated with this DAG.
DAG.residuals()	Return the residuals associated with this DAG.
DAG.residual_essential_graph()	Return the residual essential graph associated with this DAG.

graphical_models.classes.dags.dag.DAG.directed_clique_tree

DAG.**directed_clique_tree**(*verbose=False*)

Return the directed clique tree associated with this DAG.

See the following for the definition of the directed clique tree: Squires, Chandler, et al. "Active Structure Learning of Causal DAGs via Directed Clique Tree." (2020)

Parameters
verbose – if True, print out the steps taken to compute the directed clique tree.

Returns
The directed clique tree of this DAG.

Return type
networkx.MultiDiGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (1, 3), (2, 3)})
>>> dct = d.directed_clique_tree()
>>> dct.nodes
NodeView((frozenset({1, 2, 3}), frozenset({0, 1})))
>>> dct.edges
OutMultiEdgeView([(frozenset({0, 1}), frozenset({1, 2, 3}), 0)])
```

graphical_models.classes.dags.dag.DAG.contracted_directed_clique_tree

DAG.**contracted_directed_clique_tree**()

Return the contracted directed clique tree associated with this DAG.

See the following for the definition of the contracted directed clique tree: Squires, Chandler, et al. "Active Structure Learning of Causal DAGs via Directed Clique Tree." (2020)

Returns
The directed clique tree of this DAG.

Return type
networkx.MultiDiGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (1, 3), (1, 4), (3, 2), (3, 4)})
>>> cdct = d.contracted_directed_clique_tree()
>>> cdct.nodes
NodeView((frozenset({frozenset({1, 2, 3}), frozenset({1, 3, 4})}), frozenset(
↪{frozenset({0, 1})})))
>>> cdct.edges
OutEdgeView([(frozenset({frozenset({0, 1})}), frozenset({frozenset({1, 2, 3}),␣
↪frozenset({1, 3, 4})}))])
```

graphical_models.classes.dags.dag.DAG.residuals

DAG.**residuals**()

Return the residuals associated with this DAG.

See the following for the definition of residuals: Squires, Chandler, et al. "Active Structure Learning of Causal DAGs via Directed Clique Tree." (2020)

Returns
The directed clique tree of this DAG.

Return type
networkx.MultiDiGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (1, 3), (1, 4), (3, 2), (3, 4)})
>>> residuals = d.residuals()
>>> residuals
[frozenset({2, 3, 4}), frozenset({0, 1})]
```

graphical_models.classes.dags.dag.DAG.residual_essential_graph

DAG.**residual_essential_graph**()

Return the residual essential graph associated with this DAG.

See the following for the definition of the residual essential graph: Squires, Chandler, et al. "Active Structure Learning of Causal DAGs via Directed Clique Tree." (2020)

Returns
The directed clique tree of this DAG.

Return type
networkx.MultiDiGraph

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (1, 3), (1, 4), (3, 2), (3, 4)})
>>> r_eg = d.residual_essential_graph()
>>> r_eg.arcs
{(1, 2), (1, 3), (1, 4)}
```

1.2.12 Intervention Design

DAG.optimal_fully_orienting_single_node_interventions([...])	Find the smallest set of interventions which fully orients the CPDAG into this DAG.
DAG.greedy_optimal_single_node_intervention([...])	Greedily pick `num_interventions` single node interventions based on how many edges they orient.
DAG.greedy_optimal_fully_orienting_interventions([cpdag])	Find a set of interventions which fully orients a CPDAG into this DAG, using greedy selection of the interventions.

graphical_models.classes.dags.dag.DAG.optimal_fully_orienting_single_node_interventions

DAG.**optimal_fully_orienting_single_node_interventions**(*cpdag=None*, *new=False*, *verbose=False*) → Set[Hashable]

Find the smallest set of interventions which fully orients the CPDAG into this DAG.

Parameters

- **cpdag** – the starting CPDAG containing known orientations. If None, compute and use the observational essential graph.
- **new** – TODO: remove after checking that directed clique tree method works.
- **verbose** – TODO: describe.

Returns
A minimum-size set of interventions which fully orients the DAG.

Return type
interventions

Examples

```
>>> from graphical_models import DAG
>>> import itertools as itr
>>> d = DAG(arcs=set(itr.combinations(range(5), 2)))
>>> ivs = d.optimal_fully_orienting_single_node_interventions()
>>> ivs
{1, 3}
```

graphical_models.classes.dags.dag.DAG.greedy_optimal_single_node_intervention

DAG.**greedy_optimal_single_node_intervention**(*cpdag=None*, *num_interventions=1*)

Greedily pick `num_interventions` single node interventions based on how many edges they orient.

By submodularity, this will orient at least (1 - 1/e) as many edges as the optimal intervention set of size `num_interventions`.

Parameters

- **cpdag** – the starting CPDAG containing known orientations. If None, use the observational essential graph.
- **num_interventions** – the number of single-node interventions used. Default is 1.

Returns
The selected interventions and the associated cpdags that they induce.

Return type
(interventions, cpdags)

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (0, 2)})
>>> ivs, icpdags = d.greedy_optimal_single_node_intervention()
>>> ivs
[1]
>>> icpdags[0].arcs
{(0, 1), (0, 2), (1, 2)}
```

graphical_models.classes.dags.dag.DAG.greedy_optimal_fully_orienting_interventions

`DAG.greedy_optimal_fully_orienting_interventions(`*cpdag=None*`)`

Find a set of interventions which fully orients a CPDAG into this DAG, using greedy selection of the interventions. By submodularity, the number of interventions is a (1 + ln K) multiplicative approximation to the true optimal number of interventions, where K is the number of undirected edges in the CPDAG.

Parameters
cpdag – the starting CPDAG containing known orientations. If None, use the observational essential graph.

Returns
The selected interventions and the associated cpdags that they induce.

Return type
(interventions, cpdags)

Examples

```
>>> from graphical_models import DAG
>>> d = DAG(arcs={(0, 1), (1, 2), (0, 2), (0, 3), (1, 3), (2, 3)})
>>> ivs, icpdags = d.greedy_optimal_fully_orienting_interventions()
>>> ivs
[1, 2]
>>> icpdags[0].edges
{frozenset({2, 3})}
>>> icpdags[1].edges
set()
```

1.3 PDAG

1.3.1 Overview

class graphical_models.**PDAG**(*nodes: Set = {}*, *arcs: Set = {}*, *edges: Set = {}*, *known_arcs={}*, *new=False*)

1.3.2 Methods

PDAG.copy()	Return a copy of the graph
PDAG.to_amat([node_list, source_axis])	Return an adjacency matrix for the graph
PDAG.from_amat(amat[, source_axis])	Return a PDAG with arcs/edges given by amat

graphical_models.PDAG.copy

PDAG.**copy**()

Return a copy of the graph

graphical_models.PDAG.to_amat

PDAG.**to_amat**(*node_list: ~typing.Optional[list] = None*, *source_axis=0) -> (numpy.ndarray*, *<class 'list'>*)

Return an adjacency matrix for the graph

graphical_models.PDAG.from_amat

classmethod PDAG.**from_amat**(*amat: numpy.ndarray*, *source_axis=0*)

Return a PDAG with arcs/edges given by amat

Graph modification

PDAG.remove_node(node)	Remove a node from the graph

graphical_models.PDAG.remove_node

PDAG.**remove_node**(*node*)

Remove a node from the graph

Graph properties

PDAG.has_edge(i, j)	Return True if the graph contains the edge i--j
PDAG.has_edge_or_arc(i, j)	Return True if the graph contains the edge i--j or an arc i->j or i<-j

graphical_models.PDAG.has_edge

PDAG.**has_edge**(*i*, *j*)

Return True if the graph contains the edge i–j

graphical_models.PDAG.has_edge_or_arc

PDAG.**has_edge_or_arc**(*i*, *j*)

Return True if the graph contains the edge i–j or an arc i->j or i<-j

Comparison to other PDAGs

PDAG.shd(other)	Return the structural Hamming distance between this PDAG and another.

graphical_models.PDAG.shd

PDAG.**shd**(*other*)

Return the structural Hamming distance between this PDAG and another.

For each pair of nodes, the SHD is incremented by 1 if the edge type/presence between the two nodes is different

Functions for

PDAG.to_dag()	Return a DAG that is consistent with this CPDAG.
PDAG.all_dags([verbose])	Return all DAGs consistent with this PDAG

graphical_models.PDAG.to_dag

PDAG.**to_dag**()

Return a DAG that is consistent with this CPDAG.

Return type

d

Examples

TODO

graphical_models.PDAG.all_dags

`PDAG.all_dags(verbose=False)`

Return all DAGs consistent with this PDAG

1.4 GaussDAG

1.4.1 Overview

..autoclass:: GaussDAG

CHAPTER

TWO

RANDOM GRAPHS

directed_erdos(nnodes[, density, exp_nbrs, ...])	Generate random Erdos-Renyi DAG(s) on *nnodes* nodes with density *density*.
rand_weights(dag[, rand_weight_fn])	Generate a GaussDAG from a DAG, with random edge weights independently drawn from *rand_weight_fn*.

2.1 graphical_models.rand.directed_erdos

graphical_models.rand.**directed_erdos**(*nnodes*, *density=None*, *exp_nbrs=None*, *size=1*, *as_list=False*, *random_order=True*) → Union[DAG, List[DAG]]

Generate random Erdos-Renyi DAG(s) on *nnodes* nodes with density *density*.

Parameters

- **nnodes** – Number of nodes in each graph.
- **density** – Probability of any edge.
- **size** – Number of graphs.
- **as_list** – If True, always return as a list, even if only one DAG is generated.

Examples

```
>>> from graphical_models.rand import directed_erdos
>>> d = directed_erdos(5, .5)
```

2.2 graphical_models.rand.rand_weights

graphical_models.rand.**rand_weights**(*dag*, *rand_weight_fn: ~typing.Any = <function unif_away_zero>*) → GaussDAG

Generate a GaussDAG from a DAG, with random edge weights independently drawn from *rand_weight_fn*.

Parameters

- **dag** – DAG
- **rand_weight_fn** – Function to generate random weights.

Examples

```
>>> import causaldag as cd
>>> d = cd.DAG(arcs={(1, 2), (2, 3)})
>>> g = cd.rand.rand_weights(d)
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

C

D

F

G

H

I

L

M

N

O

P

R

S

T

U

V