causaldag Documentation

Chandler Squires

Jan 08, 2021

Contents

CHAPTER 1

Classes

1.1 AncestralGraph

1.1.1 Overview

class causaldag.classes.ancestral_graph.**AncestralGraph**(*nodes: Set[T] = frozenset()*, *directed: Set[T] = frozenset()*, *bidirected: Set[T] = frozenset()*, *undirected: Set[T] = frozenset()*)

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

1.1.2 Methods

AncestralGraph.copy()	Return a copy of this ancestral graph.
AncestralGraph.to_amat()	Convert the graph into an adjacency matrix.
AncestralGraph.from_amat(amat)	Create a graph from an adjacency matrix.

causaldag.classes.ancestral_graph.AncestralGraph.copy

AncestralGraph.**copy**()

Return a copy of this ancestral graph.

Returns A copy of the ancestral graph.

Return type *AncestralGraph*

causaldag.classes.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

causaldag.classes.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

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.

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph()
>>> g.add_node(1)
>>> g.add_node(2)
>>> len(g.nodes)
2
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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()
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph()
>>> g.add_directed(1, 2)
>>> g.directed
{(1, 2)}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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()
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph()
>>> g.add_bidirected(1, 2)
>>> g.bidirected
{frozenset({i, j})}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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()
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph()
>>> g.add_undirected(1, 2)
>>> g.undirected
{frozenset({i, j})}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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()
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph()
>>> g.add_nodes_from({1, 2})
>>> len(g.nodes)
2
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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()
```

causaldag.classes.ancestral_graph.AncestralGraph.remove_edges

AncestralGraph.**remove_edges**(*edges: Iterable[T_co]*, *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

```
>>> import causaldag as cd
>>> g = cd.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()
```

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`.

causaldag.classes.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
```

causaldag.classes.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

Functions for 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.parents_of(nodes, Set[Hashable]])	Return the parents of the node or set of nodes `nodes`.
AncestralGraph.children_of(i, Set[Hashable]])	Return the children of the node or set of nodes `i`.
AncestralGraph.spouses_of(nodes, Set[Hashable]])	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`.

causaldag.classes.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

causaldag.classes.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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph(directed={(1, 2), (2, 3)}, undirected={(1, 4)})
>>> g.parents_of(2)
{1}
>>> g.parents_of({2, 3})
{1, 2}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.AncestralGraph(directed={(1, 2), (2, 3)}, undirected={(1, 4)})
>>> g.children_of(1)
{2}
>>> g.children_of({1, 2})
{2, 3}
```

causaldag.classes.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

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

causaldag.classes.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

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

1.2 DAG

1.2.1 Copying

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

causaldag.classes.dag.DAG.copy

DAG.**copy**()
Return a copy of the current DAG.

causaldag.classes.dag.DAG.rename_nodes

DAG.**rename_nodes**(*name_map: Dict[KT, VT]*)
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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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, Set[Hashable]])	Return all nodes that are parents of the node or set of nodes `nodes`.
DAG.children_of(nodes, Set[Hashable]])	Return all nodes that are children of the node or set of nodes `nodes`.
DAG.neighbors_of(nodes, Set[Hashable]])	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, Set[Hashable]])	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.

causaldag.classes.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

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

(continues on next page)

(continued from previous page)

```
>>> g.parents_of(2)
{1}
>>> g.parents_of({2, 3})
{1, 2}
```

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(0,1), (0,2)})
>>> g.neighbors_of(0)
{1, 2}
>>> g.neighbors_of(2)
{0}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(0, 1), (1, 3), (2, 3), (3, 4})
>>> g.markov_blanket_of(1)
{0, 2, 3}
```

causaldag.classes.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

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

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.indegree_of(1)
0
>>> g.indegree_of(2)
2
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.outdegree_of(1)
2
>>> g.outdegree_of(3)
0
```

causaldag.classes.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

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

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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`.

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG()
>>> g.add_node(1)
>>> g.add_node(2)
>>> len(g.nodes)
2
```

causaldag.classes.dag.DAG.add_nodes_from

DAG.**add_nodes_from**(*nodes: Iterable[T_co]*)

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

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

See also:

add_node()

Examples

```
>>> import causaldag as cd
>>> g = cd.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}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2)})
>>> g.remove_node(2)
>>> g.nodes
{1}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG({1, 2})
>>> g.add_arc(1, 2)
>>> g.arcs
{(1, 2)}
```

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2)})
>>> g.remove_arc(1, 2)
>>> g.arcs
set()
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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`.

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(0,1), (0,2)})
>>> g.has_arc(0, 1)
True
>>> g.has_arc(1, 2)
False
```

causaldag.classes.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

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

causaldag.classes.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

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

causaldag.classes.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

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

causaldag.classes.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

Returns

Return type True if the arc is reversible, otherwise False.

Example

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (1, 3), (2, 3)})
>>> g.is_reversible(1, 2)
True
>>> g.is_reversible(1, 3)
False
```

causaldag.classes.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

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

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 3), (2, 3), (1, 2)})
>>> g.triples()
{frozenset({1, 3, 2})}
```

causaldag.classes.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

Returns

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.

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (2, 3)})
>>> g.topological_sort
[1, 2, 3]
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.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`.

Continued on next page

Table 10 – continued from previous page

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.

causaldag.classes.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

```
>>> import causaldag as cd
>>> g1 = cd.DAG(arcs={(1, 2), (2, 3)})
>>> g2 = cd.DAG(arcs={(2, 1), (2, 3)})
>>> g1.shd(g2)
1
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> g1 = cd.DAG(arcs={(1, 2), (2, 3)})
>>> g2 = cd.DAG(arcs={(2, 1), (2, 3)})
>>> g1.shd_skeleton(g2)
0
```

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(0, 1), (1, 2)})
>>> d2 = cd.DAG(arcs={(2, 1), (1, 0)})
>>> d3 = cd.DAG(arcs={(0, 1), (2, 1)})
>>> d4 = cd.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
```

causaldag.classes.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

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (3, 2)})
>>> other = cd.DAG(arcs={(1, 2)})
>>> g.is_minimal_imap(other)
False
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(0, 1), (1, 2)})
>>> d2 = cd.DAG(arcs={(0, 1), (2, 1), (3, 1)})
>>> d1.chickering_distance(d2)
3
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(0, 1), (1, 2)})
>>> d2 = cd.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)}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(0, 1), (1, 2)})
>>> d2 = cd.DAG(arcs={(0, 1), (2, 1)})
>>> cm = d1.confusion_matrix_skeleton(d2)
>>> cm["tpr"]
1.0
>>> d3 = cd.DAG(arcs={(0, 1), (0, 2)})
>>> 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, Hashable], B, Hashable], C, ...)	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.

causaldag.classes.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()

Returns

Return type is_dsep

Example

```
>>> import causaldag as cd
>>> g = cd.DAG(arcs={(1, 2), (3, 2)})
>>> g.dsep(1, 3)
True
>>> g.dsep(1, 3, 2)
False
```

causaldag.classes.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

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

causaldag.classes.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.

causaldag.classes.dag.DAG.local_markov_statements

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

```
>>> import causaldag as cd
>>> g = cd.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 other formats

DAG.from_amat(amat)	Return a DAG with arcs given by `amat`, i.e.
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.

causaldag.classes.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

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

causaldag.classes.dag.DAG.to_amat

DAG.**to_amat**(*node_list=None*) -> (<*class '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()

Returns

Return type (amat, node_list)

Example

```
>>> import causaldag as cd
>>> g = cd.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]
```

causaldag.classes.dag.DAG.from_nx

classmethod DAG.**from_nx**(*nx_graph: networkx.classes.digraph.DiGraph*)
Convert a networkx DiGraph into a DAG.

Parameters **nx_graph** – networkx DiGraph

Returns The graph as a DAG object.

Return type DAG

Examples

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

causaldag.classes.dag.DAG.to_nx

DAG.**to_nx**() → networkx.classes.digraph.DiGraph
Convert DAG to a networkx DiGraph.

Returns The graph as a networkx.DiGraph object.

Return type networkx.DiGraph

Examples

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

causaldag.classes.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

```
>>> import causaldag as cd
>>> 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 = cd.DAG.from_dataframe(df)
>>> d.arcs
{('a', 'b')}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.DAG(arcs={(0, 1)})
>>> d.to_dataframe()
   0  1
0  0  1
1  0  0
>>> d.to_dataframe(node_list=[1, 0])
   1  0
```

(continues on next page)

(continued from previous page)

```
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.

causaldag.classes.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

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

causaldag.classes.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 cd.AncestralGraph, the MAG resulting from marginalizing out *latent_nodes*.

Return type m

Examples

```
>>> import causaldag as cd
>>> d = cd.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()
```

causaldag.classes.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

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

causaldag.classes.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

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

(continues on next page)

(continued from previous page)

```
>>> 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`.

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(1, 0), (1, 2), (2, 0)})
>>> d2 = cd.DAG(arcs={(2, 0), (2, 1), (1, 0)})
>>> res_sinks = d1.resolved_sinks(d2)
{0}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d1 = cd.DAG(arcs={(0, 1), (1, 2)})
>>> d2 = cd.DAG(arcs={(2, 0), (2, 1), (1, 0)})
>>> sequence, moves = d1.chickering_sequence(d2)
```

(continues on next page)

(continued from previous page)

```
>>> sequence[1].arcs
{(1, 0), (1, 2)}
>>> sequence[2].arcs
{(1, 0), (1, 2), (2, 0)}
```

causaldag.classes.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.

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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)])
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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})}))])
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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})]
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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_interventions	Greedily pick `num_interventions` single node interventions based on how many edges they orient.
DAG.greedy_optimal_fully_orienting_interventions	Find a set of interventions which fully orients a CPDAG into this DAG, using greedy selection of the interventions.

causaldag.classes.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

```
>>> import causaldag as cd
>>> import itertools as itr
```

(continues on next page)

(continued from previous page)

```
>>> d = cd.DAG(arcs=set(itr.combinations(range(5), 2)))
>>> ivs = d.optimal_fully_orienting_single_node_interventions()
>>> ivs
{1, 3}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.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)}
```

causaldag.classes.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

```
>>> import causaldag as cd
>>> d = cd.DAG(arcs={(0, 1), (1, 2), (0, 2), (0, 3), (1, 3), (2, 3)})
>>> ivs, icpdags = d.greedy_optimal_fully_orienting_interventions()
```

(continues on next page)

(continued from previous page)

```
>>> ivs
[1, 2]
>>> icpdags[0].edges
{frozenset({2, 3})}
>>> icpdags[1].edges
set()
```

1.3 PDAG

1.3.1 Overview

class causaldag.classes.pdag.**PDAG**(*nodes: Set[T] = {}*, *arcs: Set[T] = {}*, *edges: Set[T] = {}*, *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

causaldag.classes.pdag.PDAG.copy

PDAG.**copy**()
Return a copy of the graph

causaldag.classes.pdag.PDAG.to_amat

PDAG.**to_amat**(*node_list: list = None*, *source_axis=0*) -> (<class 'numpy.ndarray'>, <class 'list'>)
Return an adjacency matrix for the graph

causaldag.classes.pdag.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
PDAG.add_known_arc(i, j)	

causaldag.classes.pdag.PDAG.remove_node

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

causaldag.classes.pdag.PDAG.add_known_arc

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

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

causaldag.classes.pdag.PDAG.has_edge

PDAG.**has_edge**(*i*, *j*)
Return True if the graph contains the edge i–j

causaldag.classes.pdag.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.

causaldag.classes.pdag.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

causaldag.classes.pdag.PDAG.to_dag

PDAG.**to_dag**()
Return a DAG that is consistent with this CPDAG.

Returns

Return type d

Examples

TODO

causaldag.classes.pdag.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 2

Utils

2.1 Core Utils

causaldag.utils.core_utils.**defdict2dict**(*defdict*, *keys*)

causaldag.utils.core_utils.**is_symmetric**(*matrix*, *tol=1e-08*)

causaldag.utils.core_utils.**iszero**(*a*, *atol=1e-08*)

causaldag.utils.core_utils.**ix_map_from_list**(*l*)

causaldag.utils.core_utils.**powerset**(*s: Iterable[T_co]*, *r_min=0*, *r_max=None*) → Iterable[T_co]

causaldag.utils.core_utils.**powerset_predicate**(*s: Iterable[T_co]*, *predicate: Callable[[Any], bool]*) → Iterable[T_co]

causaldag.utils.core_utils.**random_max**(*d*, *minimize=False*)

causaldag.utils.core_utils.**to_list**(*o*)

causaldag.utils.core_utils.**to_set**(*o*) → set

2.2 Conditional Independence Tests

partial_correlation_suffstat(samples[, invert])	Return the sufficient statistics for partial correlation testing.
partial_correlation_test(suffstat, VT], i, j)	Test the null hypothesis that i and j are conditionally independent given `cond_set`.
compute_partial_correlation(suffstat, i, j)	Compute the partial correlation between i and j given `cond_set`.

2.2.1 causaldag.utils.ci_tests.partial_correlation_test.partial_correlation_suffstat

causaldag.utils.ci_tests.partial_correlation_test.**partial_correlation_suffstat**(*samples*, *invert=True*) → Dict[KT, VT]

Return the sufficient statistics for partial correlation testing.

Parameters

- **samples** – (n x p) matrix, where n is the number of samples and p is the number of variables.
- **invert** – if True, compute the inverse correlation matrix, and normalize it into the partial correlation matrix. This will generally speed up the gauss_ci_test if large conditioning sets are used.

Returns dictionary of sufficient statistics

Return type dict

2.2.2 causaldag.utils.ci_tests.partial_correlation_test.partial_correlation_test

causaldag.utils.ci_tests.partial_correlation_test.**partial_correlation_test**(*suffstat: Dict[KT, VT]*, *i*, *j*, *cond_set=None*, *alpha=None*)

Test the null hypothesis that i and j are conditionally independent given `cond_set`.

Uses Fisher's z-transform.

Parameters

- **suffstat** – dictionary containing:
  - `n` – number of samples
  - `C` – correlation matrix
  - `K` (optional) – inverse correlation matrix
  - `rho` (optional) – partial correlation matrix (K, normalized so diagonals are 1).
- **i** – position of first variable in correlation matrix.
- **j** – position of second variable in correlation matrix.
- **cond_set** – positions of conditioning set in correlation matrix.
- **alpha** – Significance level.

Returns

dictionary containing:

- `statistic`

- p_value
- reject

Return type dict

2.2.3 causaldag.utils.ci_tests.partial_correlation_test.compute_partial_correlation

causaldag.utils.ci_tests.partial_correlation_test.**compute_partial_correlation**(*suffstat*, *i*, *j*, *cond_set=None*)

Compute the partial correlation between i and j given cond_set.

Parameters

- **suffstat** – dictionary containing: 'n' – number of samples 'C' – correlation matrix 'K' (optional) – inverse correlation matrix 'rho' (optional) – partial correlation matrix (K, normalized so diagonals are 1).
- **i** – position of first variable in correlation matrix.
- **j** – position of second variable in correlation matrix.
- **cond_set** – positions of conditioning set in correlation matrix.

Returns partial correlation

Return type float

2.3 Conditional Invariance Tests

gauss_invariance_suffstat(obs_samples, ...)	Helper function to compute the sufficient statistics for the gauss_invariance_test from data.
gauss_invariance_test(suffstat, context, i, ...)	Test the null hypothesis that two Gaussian distributions are equal.

2.3.1 causaldag.utils.invariance_tests.gauss_invariance.gauss_invariance_suffstat

causaldag.utils.invariance_tests.gauss_invariance.**gauss_invariance_suffstat**(*obs_samples*, *context_samples_list*)

Helper function to compute the sufficient statistics for the gauss_invariance_test from data.

Parameters

- **obs_samples** – (n x p) matrix, where n is the number of samples and p is the number of variables.
- **context_samples_list** – list of (n x p) matrices, one for each context besides observational

Returns dictionary of sufficient statistics

Return type dict

2.3.2 causaldag.utils.invariance_tests.gauss_invariance.gauss_invariance_test

causaldag.utils.invariance_tests.gauss_invariance.**gauss_invariance_test**(*suffstat*, *context*, *i: int*, *cond_set: Union[List[int], int, None] = None*, *alpha: float = 0.05*, *zero_mean=False*, *same_coeffs=False*)

Test the null hypothesis that two Gaussian distributions are equal.

Parameters

- **suffstat** – dictionary containing:
  - obs – number of samples
  - G – Gram matrix
  - contexts
- **context** – which context to test.
- **i** – position of marginal distribution.
- **cond_set** – positions of conditioning set in correlation matrix.
- **alpha** – Significance level.
- **zero_mean** – If True, assume that the regression residual has zero mean.
- **same_coeffs** – If True, assume that the regression coefficients have not changed.

Returns dictionary containing ttest_stat, ftest_stat, f_pvalue, t_pvalue, and reject.

Return type dict

2.4 Scores

local_gaussian_bic_score(node, suffstat)	parents,	
local_gaussian_bge_score(node, suffstat)	parents,	Compute the BGE score of a node given its parents.

2.4.1 causaldag.utils.scores.local_gaussian_bic_score

causaldag.utils.scores.**local_gaussian_bic_score**(*node*, *parents*, *suffstat*, *lambda_=None*)

2.4.2 causaldag.utils.scores.local_gaussian_bge_score

causaldag.utils.scores.**local_gaussian_bge_score**(*node*, *parents*, *suffstat*, *alpha_mu=None*, *alpha_w=None*, *inverse_scale_matrix=None*, *parameter_mean=None*, *is_diagonal=True*)

Compute the BGE score of a node given its parents.

Parameters

- **node** – TODO - describe.
- **parents** – TODO - describe.
- **suffstat** – dictionary containing:
  - n – number of samples
  - S – sample covariance matrix
  - mu – sample mean
- **alpha_mu** – TODO - describe. Default is the number of variables.
- **alpha_w** – TODO - describe. Default is the (number of variables) + alpha_mu + 1
- **inverse_scale_matrix** – TODO - describe. Default is the identity matrix.
- **parameter_mean** – TODO - describe. Default is the zero vector.
- **is_diagonal** – TODO - describe.

Returns BGE score.

Return type float

CHAPTER 3

Structure Learning

3.1 Undirected Structure Learning

partial_correlation_threshold	
threshold_ug(nodes, ci_tester)	Estimate an undirected graph by testing whether each pair of nodes is independent given all others.

3.1.1 causaldag.structure_learning.threshold_ug

causaldag.structure_learning.**threshold_ug**(*nodes: set*, *ci_tester: causaldag.utils.ci_tests.ci_tester.CI_Tester*) → causaldag.classes.undirected_graph.UndirectedGraph

Estimate an undirected graph by testing whether each pair of nodes is independent given all others.

Parameters

- **nodes** – Nodes in the graph.
- **ci_tester** – Conditional independence tester.

Examples

TODO

3.2 Covariance Structure Learning

covariance_graph_gauss(ci_tester)	Estimate a covariance graph by testing whether each pair of nodes is independent, which reduces to thresholding correlations (after the Fisher z-transform) for multivariate Gaussian data.

3.2.1 causaldag.structure_learning.covariance_graph_gauss

causaldag.structure_learning.**covariance_graph_gauss**(*ci_tester*)

Estimate a covariance graph by testing whether each pair of nodes is independent, which reduces to thresholding correlations (after the Fisher z-transform) for multivariate Gaussian data.

Parameters **ci_tester** – Conditional independence tester.

Examples

TODO

3.3 DAG Structure Learning

permutation2dag(perm, ci_tester[, verbose, ...])	Estimate the minimal IMAP of a DAG which is consistent with the given permutation.
sparsest_permutation(nodes, ci_tester[, ...])	Estimate the Markov equivalence class of a DAG using the Sparsest Permutations (SP) algorithm.
gsp(nodes, ci_tester, depth, nruns, verbose, ...)	Estimate the Markov equivalence class of a DAG using the Greedy Sparsest Permutations (GSP) algorithm.

3.3.1 causaldag.structure_learning.permutation2dag

causaldag.structure_learning.**permutation2dag**(*perm: list*, *ci_tester: causaldag.utils.ci_tests.ci_tester.CI_Tester*, *verbose=False*, *fixed_adjacencies: Set[FrozenSet[Hashable]] = {}*, *fixed_gaps: Set[FrozenSet[Hashable]] = {}*, *progress=False*)

Estimate the minimal IMAP of a DAG which is consistent with the given permutation.

Parameters

- **perm** – list of nodes representing the permutation.
- **ci_tester** – object for testing conditional independence.
- **verbose** – if True, log each CI test.
- **fixed_adjacencies** – set of nodes known to be adjacent.
- **fixed_gaps** – set of nodes known not to be adjacent.

Examples

```
>>> from causaldag.utils.ci_tests import MemoizedCI_Tester, partial_correlation_
↪test, partial_correlation_suffstat
>>> perm = [0,1,2]
>>> suffstat = partial_correlation_suffstat(samples)
>>> ci_tester = MemoizedCI_Tester(partial_correlation_test, suffstat)
>>> permutation2dag(perm, ci_tester, fixed_gaps={frozenset({1, 2})})
```

3.3.2 causaldag.structure_learning.sparsest_permutation

causaldag.structure_learning.**sparsest_permutation**(*nodes*, *ci_tester*, *progress=False*)

Estimate the Markov equivalence class of a DAG using the Sparsest Permutations (SP) algorithm.

Parameters

- **nodes** – list of nodes.
- **ci_tester** – object for testing conditional independence.
- **progress** – if True, show a progress bar over the enumeration of permutations.

Examples

```
>>> from causaldag.utils.ci_tests import MemoizedCI_Tester, partial_correlation_
↪test, partial_correlation_suffstat
>>> import causaldag as cd
>>> import random
>>> import numpy as np
>>> random.seed(1212)
>>> np.random.seed(12131)
>>> nnodes = 7
>>> d = cd.rand.directed_erdos(nnodes, exp_nbrs=2)
>>> g = cd.rand.rand_weights(d)
>>> samples = g.sample(1000)
>>> suffstat = partial_correlation_suffstat(samples)
>>> ci_tester = MemoizedCI_Tester(partial_correlation_test, suffstat, alpha=1e-3)
>>> est_dag = cd.sparsest_permutation(set(range(nnodes)), ci_tester,␣
↪progress=True)
>>> true_cpdag = d.cpdag()
>>> est_cpdag = est_dag.cpdag()
>>> print(true_cpdag.shd(est_cpdag))
>>> 0
```

3.3.3 causaldag.structure_learning.gsp

causaldag.structure_learning.**gsp**(*nodes: set, ci_tester: causaldag.utils.ci_tests.ci_tester.CI_Tester, depth: Optional[int] = 4, nruns: int = 5, verbose: bool = False, initial_undirected: Union[str, causaldag.classes.undirected_graph.UndirectedGraph, None] = 'threshold', initial_permutations: Optional[List[T]] = None, fixed_orders={}, fixed_adjacencies={}, fixed_gaps={}, use_lowest=True, max_iters=inf, factor=2, progress_bar=False, summarize=False) -> (<class 'causaldag.classes.dag.DAG'>, typing.List[typing.List[typing.Dict]]*)

Estimate the Markov equivalence class of a DAG using the Greedy Sparsest Permutations (GSP) algorithm.

Parameters

- **nodes** – Labels of nodes in the graph.
- **ci_tester** – A conditional independence tester, which has a method is_ci taking two sets A and B, and a conditioning set C, and returns True/False.
- **depth** – Maximum depth in depth-first search. Use None for infinite search depth.
- **nruns** – Number of runs of the algorithm. Each run starts at a random permutation and the sparsest DAG from all runs is returned.
- **verbose** – TODO
- **initial_undirected** – Option to find the starting permutation by using the minimum degree algorithm on an undirected graph that is Markov to the data. You can provide the undirected graph yourself, use the default 'threshold' to do simple thresholding on the partial correlation matrix, or select 'None' to start at a random permutation.
- **initial_permutations** – A list of initial permutations with which to start the algorithm. This option is helpful when there is background knowledge on orders. This option is mutually exclusive with initial_undirected.
- **fixed_orders** – Tuples (i, j) where i is known to come before j.
- **fixed_adjacencies** – Tuples (i, j) where i and j are known to be adjacent.
- **fixed_gaps** – Tuples (i, j) where i and j are known to be non-adjacent.

See also:

`pcalg(), igsp(), unknown_target_igsp()`

Returns

Return type (est_dag, summaries)

CHAPTER 4

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*.

4.1 causaldag.rand.directed_erdos

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

```
>>> import causaldag as cd
>>> d = cd.rand.directed_erdos(5, .5)
```

4.2 causaldag.rand.rand_weights

causaldag.rand.**rand_weights**(*dag*, *rand_weight_fn: Any = <function unif_away_zero>*) → causaldag.classes.gaussdag.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 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

G

H

I

L

M

N

O

P

R

S

T

U

V