

API Documentation

API Documentation

March 28, 2014

Contents

| | |
|---|-----------|
| Contents | 1 |
| 1 Package coinor.gimpy | 2 |
| 1.1 Modules | 2 |
| 1.2 Variables | 2 |
| 2 Module coinor.gimpy.global_constants | 4 |
| 2.1 Functions | 4 |
| 2.2 Variables | 4 |
| 2.3 Class MultipleNodeException | 5 |
| 2.3.1 Methods | 5 |
| 2.3.2 Properties | 5 |
| 2.4 Class MultipleEdgeException | 6 |
| 2.4.1 Methods | 6 |
| 2.4.2 Properties | 6 |
| 3 Module coinor.gimpy.graph | 7 |
| 3.1 Variables | 8 |
| 3.2 Class Node | 9 |
| 3.2.1 Methods | 9 |
| 3.2.2 Properties | 11 |
| 3.3 Class Graph | 11 |
| 3.3.1 Methods | 11 |
| 3.3.2 Properties | 47 |
| 3.4 Class DisjointSet | 47 |
| 3.4.1 Methods | 48 |
| 3.4.2 Properties | 49 |
| 4 Module coinor.gimpy.tree | 50 |
| 4.1 Variables | 50 |
| 4.2 Class Tree | 51 |
| 4.2.1 Methods | 51 |
| 4.2.2 Properties | 55 |
| 4.3 Class BinaryTree | 55 |
| 4.3.1 Methods | 56 |
| 4.3.2 Properties | 60 |

1 Package coinor.gimpy

1.1 Modules

- **global_constants:** This file has global constants required for GIMPy.
(Section 2, p. 4)
- **graph:** A Graph class implementation.
(Section 3, p. 7)
- **tree:** Tree class built on top of Graph class.
(Section 4, p. 50)

1.2 Variables

| Name | Description |
|------------------------------|--|
| CLUSTER_ATTRIBUTES | Value: set(['K', 'URL', 'bgcolor', 'color', 'colorscheme', 'fill...]) |
| DEFAULT_EDGE_ATTRIBUTES | Value: {} |
| DEFAULT_GRAPH_ATTRIBUTES | Value: {} |
| DEFAULT_NODE_ATTRIBUTES | Value: {} |
| DIRECTED_GRAPH | Value: 'digraph' |
| DOT2TEX_INSTALLED | Value: True |
| DOT2TEX_TEMPLATE | Value: '\n\\documentclass[landscape]{article}\n\\usepackage{x11n...] |
| DOT_KEYWORDS | Value: ['graph', 'subgraph', 'digraph', 'node', 'edge', 'strict'] |
| EDGE_ATTRIBUTES | Value: set(['URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color...]) |
| EDGE.CONNECT_SYMBOL | Value: {'digraph': ' -> ', 'graph': ' -- '} |
| ETREE_INSTALLED | Value: True |
| GRAPH_ATTRIBUTES | Value: set(['Damping', 'K', 'URL', 'aspect', 'bb', 'bgcolor', 'c...]) |
| ID.RE.ALPHA.NUMS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_]*\$') |
| ID.RE.ALPHA.NUMS.WITH._PORTS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_:"]*[a-zA-Z0-9_,"]...] |
| ID.RE.DBL.QUOTED | Value: re.compile(r'(?su)^"\$.*\$') |
| ID.RE.HTML | Value: re.compile(r'(?su)^<.*>\$') |
| ID.RE.NUM | Value: re.compile(r'(?u)^[0-9,]+\$') |
| ID.RE.WITH.PORT | Value: re.compile(r'(?u)^([~:]*)([~:]*)\$') |
| INF | Value: 10000 |
| NODE_ATTRIBUTES | Value: set(['URL', 'color', 'colorscheme', 'comment', 'distortio...]) |
| PIL_INSTALLED | Value: True |
| PYGAME_INSTALLED | Value: True |
| UNDIRECTED_GRAPH | Value: 'graph' |
| XDOT_INSTALLED | Value: True |

continued on next page

| Name | Description |
|-------------|------------------------------|
| --package-- | Value: 'coinor.gimpy' |

2 Module `coinor.gimpy.global_constants`

This file has global constants required for GIMPy.

2.1 Functions

| |
|---|
| <code>quote_if_necessary(<i>s</i>)</code> |
|---|

| |
|-------------------------------------|
| <code>needs_quotes(<i>s</i>)</code> |
|-------------------------------------|

| |
|---|
| Checks whether a string is a dot language ID. It will check whether the string is solely composed by the characters allowed in an ID or not. If the string is one of the reserved keywords it will need quotes too but the user will need to add them manually. |
|---|

2.2 Variables

| Name | Description |
|-----------------------------|---|
| GRAPH_ATTRIBUTES | Value: set(['Damping', 'K', 'URL', 'aspect', 'bb', 'bgcolor', 'c...']) |
| DEFAULT_GRAPH_ATTRIBUTES | Value: {} |
| EDGE_ATTRIBUTES | Value: set(['URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color...']) |
| DEFAULT_EDGE_ATTRIBUTES | Value: {} |
| NODE_ATTRIBUTES | Value: set(['URL', 'color', 'colorscheme', 'comment', 'distortio...']) |
| DEFAULT_NODE_ATTRIBUTES | Value: {} |
| CLUSTER_ATTRIBUTES | Value: set(['K', 'URL', 'bgcolor', 'color', 'colorscheme', 'fill...']) |
| DIRECTED_GRAPH | Value: 'digraph' |
| UNDIRECTED_GRAPH | Value: 'graph' |
| EDGE_CONNECT_SYMBOL | Value: {'digraph': ' -> ', 'graph': ' -- '} |
| PYGAME_INSTALLED | Value: None |
| DOT2TEX_INSTALLED | Value: None |
| PIL_INSTALLED | Value: None |
| XDOT_INSTALLED | Value: None |
| ETREE_INSTALLED | Value: None |
| INF | Value: 10000 |
| DOT2TEX_TEMPLATE | Value: '\n\\documentclass[landscape]{article}\n\\usepackage{x11n...' |
| DOT_KEYWORDS | Value: ['graph', 'subgraph', 'digraph', 'node', 'edge', 'strict'] |
| ID_RE_ALPHA_NUMS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_]*\$') |
| ID_RE_ALPHA_NUMS_WITH_PORTS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_:.]*[a-zA-Z0-9_]"...') |

continued on next page

| Name | Description |
|-------------------------------|--|
| <code>ID_RE_NUM</code> | Value: <code>re.compile(r'(?u)^[0-9,]+\$')</code> |
| <code>ID_RE_WITH_PORT</code> | Value: <code>re.compile(r'(?u)^([[::]]*):([[::]]*)\$')</code> |
| <code>ID_RE_DBL_QUOTED</code> | Value: <code>re.compile(r'(?su)^".*"\$')</code> |
| <code>ID_RE_HTML</code> | Value: <code>re.compile(r'(?su)^<.*>\$')</code> |
| <code>__package__</code> | Value: <code>'coinor.gimpy'</code> |

2.3 Class `MultipleNodeException`



2.3.1 Methods

Inherited from `exceptions.Exception`

`__init__()`, `__new__()`

Inherited from `exceptions.BaseException`

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,
`__setattr__()`, `__setstate__()`, `__str__()`, `__unicode__()`

Inherited from `object`

`__format__()`, `__hash__()`, `__reduce_ex__()`, `__sizeof__()`, `__subclasshook__()`

2.3.2 Properties

| Name | Description |
|---|-------------|
| <i>Inherited from <code>exceptions.BaseException</code></i> | |
| <code>args</code> , <code>message</code> | |
| <i>Inherited from <code>object</code></i> | |
| <code>__class__</code> | |

2.4 Class `MultipleEdgeException`



2.4.1 Methods

Inherited from `exceptions.Exception`

`__init__()`, `__new__()`

Inherited from `exceptions.BaseException`

`__delattr__()`, `__getattr__()`, `__getitem__()`, `__getslice__()`, `__reduce__()`, `__repr__()`,
`__setattr__()`, `__setstate__()`, `__str__()`, `__unicode__()`

Inherited from `object`

`__format__()`, `__hash__()`, `__reduce_ex__()`, `__sizeof__()`, `__subclasshook__()`

2.4.2 Properties

| Name | Description |
|---|-------------|
| <i>Inherited from <code>exceptions.BaseException</code></i> | |
| <code>args</code> , <code>message</code> | |
| <i>Inherited from <code>object</code></i> | |
| <code>__class__</code> | |

3 Module `coinor.gimpy.graph`

A Graph class implementation. The aim for this implementation is

1. To reflect implementation methods in literature as much as possible
3. To have something close to a "classic" object-oriented design (compared to previous versions)

This implementation can be considered as a compromise between a graph class designed for visualization and an efficient graph data structure.

One deviation from standard Graph implementations is to keep in neighbors in an other adjacency list. We do this for efficiency reasons considering traversing residual graphs.

We have a class for Graph and a class for Node. Edges are not represented as objects. They are kept in a dictionary which also keeps their attributes.

Graph display related methods are inspired from Pydot. They are re-written considering GIMPy needs. We also borrow two methods from Pydot, see `global_constants.py` for details.

Default graph type is an undirected graph.

No custom exception will raise when the user tries to get `inneighbors` of an undirected graph. She should be aware of this. Python will raise an exception since user is trying to read an attribute that does not exists.

Methods that implement algorithms has `display` argument in their API. If this argument is not specified global display setting will be used for display purposes of the algorithm method implements. You can use `display` argument to get visualization of algorithm without changing global display behavior of your Graph/Tree object.

Method documentation strings are orginized as follows.

API: `method_name(arguments)`

Description: Description of the method.

Input: Arguments and their explanation.

Pre: Necessary class attributes that should exists, methods to be called before this method.

Post: Class attributes changed within the method.

Return: Return value of the method.

TODO(aykut):

```

-> svg display mode
-> label_strong_components() API change. Check backward compatibilty.
-> dfs should use search()?
-> display mode svg is not supported.
future:
-> The solution we find is not strongly feasible. Fix this.

```

Version: 1.1.1

Author: Ted Ralphs, Aykut Bulut (ted@lehigh.edu, aykut@lehigh.edu)

License: BSD

3.1 Variables

| Name | Description |
|---------------------------------------|---|
| <code>__maintainer__</code> | Value: 'Aykut Bulut' |
| <code>__email__</code> | Value: 'aykut@lehigh.edu' |
| <code>__url__</code> | Value: None |
| <code>__title__</code> | Value: 'Linked list data structure' |
| <code>PYGAME_INSTALLED</code> | Value: True |
| <code>DOT2TEX_INSTALLED</code> | Value: True |
| <code>PIL_INSTALLED</code> | Value: True |
| <code>XDOT_INSTALLED</code> | Value: True |
| <code>ETREE_INSTALLED</code> | Value: True |
| <code>CLUSTER_ATTRIBUTES</code> | Value: set(['K', 'URL', 'bgcolor', 'color', 'colorscheme', 'fill...']) |
| <code>DEFAULT_EDGE_ATTRIBUTES</code> | Value: {} |
| <code>DEFAULT_GRAPH_ATTRIBUTES</code> | Value: {} |
| <code>DEFAULT_NODE_ATTRIBUTES</code> | Value: {} |
| <code>DIRECTED_GRAPH</code> | Value: 'digraph' |
| <code>DOT2TEX_TEMPLATE</code> | Value: '\n\\documentclass[landscape]{article}\n\\usepackage[x11n...] |
| <code>DOT_KEYWORDS</code> | Value: ['graph', 'subgraph', 'digraph', 'node', 'edge', 'strict'] |
| <code>EDGE_ATTRIBUTES</code> | Value: set(['URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color...']) |
| <code>EDGE_CONNECT_SYMBOL</code> | Value: {'digraph': ' -> ', 'graph': ' -- '} |
| <code>GRAPH_ATTRIBUTES</code> | Value: set(['Damping', 'K', 'URL', 'aspect', 'bb', 'bgcolor', 'c...']) |

continued on next page

| Name | Description |
|-----------------------------|--|
| ID_RE_ALPHA_NUMS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_]*\$') |
| ID_RE_ALPHA_NUMS_WITH_PORTS | Value: re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_:"]*[a-zA-Z0-9_,"].. |
| ID_RE_DBL_QUOTED | Value: re.compile(r'(?su)^"\$.*\$') |
| ID_RE_HTML | Value: re.compile(r'(?su)^<.*>\$') |
| ID_RE_NUM | Value: re.compile(r'(?u)^[0-9,]+\$') |
| ID_RE_WITH_PORT | Value: re.compile(r'(?u)^([~:]*):([~:]*)\$') |
| INF | Value: 10000 |
| NODE_ATTRIBUTES | Value: set(['URL', 'color', 'colorscheme', 'comment', 'distortio... |
| UNDIRECTED_GRAPH | Value: 'graph' |
| __package__ | Value: 'coinor.gimpy' |

3.2 Class Node

object └─
 coinor.gimpy.graph.Node

Node class. A node object keeps node attributes. Has a method to write node in Dot language grammar.

3.2.1 Methods

```
__init__(self, name, **attr)
```

API: `__init__(self, name, **attrs)`
Description:
Node class constructor. Sets name and attributes using arguments.
Input:
 name: Name of node.
 **attrs: Node attributes.
Post:
 Sets `self.name` and `self.attr`.
Overrides: `object.__init__`

get_attr(*self*, *attr*)API: get_attr(*self*, *attr*)

Description:

Returns node attribute *attr*.

Input:

attr: Node attribute to get.

Return:

Returns Node attribute *attr* if exists returns None, otherwise.

set_attr(*self*, *attr*, *value*)API: set_attr(*self*, *attr*, *value*)

Description:

Sets node attribute *attr* to *value*.

Input:

attr: Node attribute to set.*value*: New value of the attribute.

Post:

Updates *self.attr[attr]*.

to_string(*self*)API: to_string(*self*)

Description:

Returns string representation of node in dot language.

Return:

String representation of node.

__repr__(*self*)API: __repr__(*self*)

Description:

Returns string representation of node in dot language.

Return:

String representation of node.

Overrides: object.__repr__

Inherited from object

`__delattr__()`, `__format__()`, `__getattr__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`,
`__setattr__()`, `__sizeof__()`, `__str__()`, `__subclasshook__()`

3.2.2 Properties

| Name | Description |
|------------------------------|-------------|
| <i>Inherited from object</i> | |
| <code>__class__</code> | |

3.3 Class Graph

object └─
 coinor.gimpy.graph.Graph

Known Subclasses: `coinor.gimpy.graph.DisjointSet`, `coinor.gimpy.tree.Tree`

Graph class, implemented using adjacency list. See GIMPY README for more information.

3.3.1 Methods

| |
|--|
| <code>__init__(self, **attr)</code> |
| <p>API: <code>__init__(self, **attrs)</code></p> <p>Description: Graph class constructor. Sets attributes using argument.</p> <p>Input: <code>**attrs</code>: Graph attributes.</p> <p>Post: Sets following attributes using <code>**attrs</code>; <code>self.attr</code>, <code>self.graph_type</code>. Creates following initial attributes; <code>self.neighbors</code>, <code>self.in_neighbors</code>, <code>self.nodes</code>, <code>self.out_neighbors</code>, <code>self.cluster</code></p> <p>Overrides: <code>object.__init__</code></p> |

`--repr--(self)`

API: `--repr--(self)`

Description:

Returns string representation of the graph.

Return:

String representation of the graph.

Overrides: `object.__repr__`

`add_edge(self, name1, name2, **attr)`

API: `add_edge(self, name1, name2, **attr)`

Description:

Adds edge to the graph. Sets edge attributes using `attr` argument.

Input:

`name1`: Name of the source node (if directed).`name2`: Name of the sink node (if directed).`attr`: Edge attributes.

Pre:

Graph should not already contain this edge. We do not allow multiple edges with same source and sink nodes.

Post:

`self.edge_attr` is updated.`self.neighbors`, `self.nodes` and `self.in_neighbors` are updated if graph was missing at least one of the nodes.

add_node(*self*, *name*, ****attr**)API: add_node(*self*, *name*, ****attr**)

Description:

Adds node to the graph.

Pre:

Graph should not contain a node with this name. We do not allow multiple nodes with the same name.

Input:

name: Name of the node.

attr: Node attributes.

Post:

self.neighbors, self.nodes and self.in_neighbors are updated.

Return:

Node (a Node class instance) added to the graph.

augment_cycle(*self*, *amount*, *cycle*)

API:

augment_cycle(*self*, *amount*, *cycle*):

Description:

Augments 'amount' unit of flow along cycle.

Pre:

Arcs should have 'flow' attribute.

Inputs:

amount: An integer representing the amount to augment

cycle: A list representing a cycle

Post:

Changes 'flow' attributes of arcs.

bfs(*self*, *root*, *display*=None, *component*=None)API: bfs(*self*, *root*, *display* = None, *component*=None)

Description:

Make a breadth-first search starting from node with name *root*.

Input:

root: Starting node name.*display*: display method.*component*: component number.

Post:

Nodes will have 'component' attribute that will have component number as value.

check_edge(*self*, *name1*, *name2*)API: check_edge(*self*, *name1*, *name2*)

Description:

Return True if edge exists, False otherwise.

Input:

name1: name of the source node.*name2*: name of the sink node.

Return:

Returns True if edge exists, False otherwise.

create(*self*, *layout*, *format*, ***args*)

API:

create(*self*, *layout*, *format*, ***args*)

Description:

Returns postscript representation of graph.

Input:

layout: Dot layout for generating graph image.*format*: Image format, all format supported by Dot are wellcome.

Return:

Returns postscript representation of graph.

create_cluster(*self*, *node_list*, *cluster_attrs*={}, *node_attrs*={})

API:

`create_cluster(self, node_list, cluster_attrs, node_attrs)`

Description:

Creates a cluster from the node given in the node list.

Input:

node_list: List of nodes in the cluster.*cluster_attrs*: Dictionary of cluster attributes, see Dot language grammar documentation for details.*node_attrs*: Dictionary of node attributes. It will overwrite previous attributes of the nodes in the cluster.

Post:

A cluster will be created. Attributes of the nodes in the cluster may change.

create_residual_graph(*self*)API: `create_residual_graph(self)`

Description:

Creates and returns residual graph, which is a Graph instance itself.

Pre:

(1) Arcs should have 'flow', 'capacity' and 'cost' attribute

(2) Graph should be a directed graph

Return:

Returns residual graph, which is a Graph instance.

cycle_canceling(*self*, *display*)

API:

`cycle_canceling(self, display)`

Description:

Solves minimum cost feasible flow problem using cycle canceling algorithm. Returns True when an optimal solution is found, returns False otherwise. 'flow' attribute values of arcs should be considered as junk when returned False.

Input:

display: Display method.

Pre:

- (1) Arcs should have 'capacity' and 'cost' attribute.
- (2) Nodes should have 'demand' attribute, this value should be positive if the node is a supply node, negative if it is demand node and 0 if it is transshipment node.
- (3) graph should not have node 's' and 't'.

Post:

Changes 'flow' attributes of arcs.

Return:

Returns True when an optimal solution is found, returns False otherwise.

del_edge(*self*, *e*)API: `del_edge(self, e)`

Description:

Removes edge from graph.

Input:

e: Tuple that represents edge, in (source,sink) form.

Pre:

Graph should contain this edge.

Post:

self.edge_attr, self.neighbors and self.in_neighbors are updated.

del_node(*self*, *name*)API: del_node(*self*, *name*)

Description:

Removes node from Graph.

Input:

name: Name of the node.

Pre:

Graph should contain a node with this name.

Post:

self.neighbors, *self*.nodes and *self*.in_neighbors are updated.**dfs**(*self*, *root*, *disc_count*=0, *finish_count*=1, *component*=None, *transpose*=False, *display*=None, *pred*=None)API: dfs(*self*, *root*, *disc_count* = 0, *finish_count* = 1, *component*=None, *transpose*=False)

Description:

Make a depth-first search starting from node with name *root*.

Input:

root: Starting node name. *disc_count*: Discovery time. *finish_count*: Finishing time. *component*: component number. *transpose*: Goes in the reverse direction along edges if *transpose* is True.

Post:

Nodes will have 'component' attribute that will have component number as value. Updates 'disc_time' and 'finish_time' attributes of nodes which represents discovery time and finishing time.

Return:

Returns a tuple that has discovery time and finish time of the last node in the following form (disc_time, finish_time).

```
display(self, highlight=None, basename='graph', format='png',
        pause=True)
```

API:

```
display(self, highlight = None, basename = 'graph', format = 'png',
        pause = True)
```

Description:

Displays graph according to the arguments provided.

Current display modes: 'off', 'file', 'pygame', 'PIL', 'xdot', 'svg'

Current layout modes: Layouts provided by graphviz ('dot', 'fdp', 'circo', etc.) and 'dot2tex'.

Current formats: Formats provided by graphviz ('ps', 'pdf', 'png', etc.)

Input:

highlight: List of nodes to be highlighted.

basename: File name. It will be used if display mode is 'file'.

format: Image format, all format supported by Dot are wellcome.

pause: If display is 'pygame' and pause is True pygame will pause and wait for user input before closing the display. It will close display window straightaway otherwise.

Post:

A display window will pop up or a file will be written depending on display mode.

```
edge_to_string(self, e)
```

API: edge_to_string(self, e)

Description:

Return string that represents edge e in dot language.

Input:

e: Edge tuple in (source,sink) format.

Pre:

Graph should have this edge.

Return:

String that represents given edge.

fifo_label_correcting(*self*, *source*)

API:

`fifo_label_correcting(self, source)`

Description:

finds shortest path from source to every other node. Returns predecessor dictionary. If graph has a negative cycle, detects it and returns to it.

Pre:

(1) 'cost' attribute of arcs. It will be used to compute shortest path.

Input:

source: source node

Post:

Modifies 'distance' attribute of nodes.

Return:

If there is no negative cycle returns to (True, pred), otherwise returns to (False, cycle) where pred is the predecessor dictionary and cycle is a list of nodes that represents cycle. It is in [n₁, n₂, ..., n_k] form where the cycle has k nodes.

find_cycle_capacity(*self*, *cycle*)

API:

`find_cycle_capacity(self, cycle):`

Description:

Finds capacity of the cycle input.

Pre:

(1) Arcs should have 'capacity' attribute.

Input:

cycle: a list representing a cycle

Return:

Returns an integer number representing capacity of cycle.

find_feasible_flow(*self*)

API:

```
find_feasible_flow(self)
```

Description:

Solves feasible flow problem, stores solution in 'flow' attribute or arcs. This method is used to get an initial feasible flow for simplex and cycle canceling algorithms. Uses max_flow() method. Other max flow methods can also be used. Returns True if a feasible flow is found, returns False, if the problem is infeasible. When the problem is infeasible 'flow' attributes of arcs should be considered as junk.

Pre:

- (1) 'capacity' attribute of arcs
- (2) 'demand' attribute of nodes

Post:

Keeps solution in 'flow' attribute of arcs.

Return:

Returns True if a feasible flow is found, returns False, if the problem is infeasible

get_approximate_diameter(*self*)

API:

```
get_approximate_diameter(self)
```

Description:

Returns an approximation of the diameter of the graph, which is accurate in many cases. Diameter is defined as follows.

distance(n,m): shortest unweighted path from n to m

eccentricity(n) = $\max_m \text{distance}(n,m)$

diameter = $\max_n \text{eccentricity}(n) = \max_n \max_m \text{distance}(n,m)$

Return:

Returns approximate diameter of the graph.

get_degree(*self*)

API:

`get_degree(self)`

Description:

Returns degrees of nodes in dictionary format.

Return:

Returns a dictionary of node degrees. Keys are node names, values are corresponding degrees.

get_diameter(*self*)

API:

`get_diameter(self)`

Description:

Returns diameter of the graph. Diameter is defined as follows.

`distance(n,m): shortest unweighted path from n to m``eccentricity(n) = $\max_m \text{distance}(n,m)$` `diameter = $\min_n \text{eccentricity}(n) = \min_n \max_m \text{distance}(n,m)$`

Return:

Returns diameter of the graph.

get_edge_attr(*self*, *n*, *m*, *attr*)API: `get_edge_attr(self, n, m, attr)`

Description:

Returns attribute attr of edge (n,m).

Input:

`n: Source node name.``m: Sink node name.``attr: Attribute of edge.`

Pre:

Graph should have this edge.

Return:

Value of edge attribute attr.

get_edge_cost(*self*, *edge*)

API: get_edge_cost(*self*, *edge*)

Description:

Returns cost attr of edge, required for minimum_spanning_tree_kruskal().

Input:

edge: Tuple that represents edge, in (source,sink) form.

Return:

Returns cost attribute value of the edge.

get_edge_list(*self*)

API: get_edge_list(*self*)

Description:

Returns edge list.

Return:

List of edges, edges are tuples and in (source,sink) format.

get_edge_num(*self*)

API: get_edge_num(*self*)

Description:

Returns number of edges.

Return:

Number of edges.

get_in_neighbors(*self*, *name*)

API: get_in_neighbors(*self*, *name*)

Description:

Returns list of in neighbors of given node.

Input:

name: Node name.

Pre:

Graph should have this node.

Return:

List of in-neighbor node names.

get_layout(*self*)

API:

`get_layout(self)`

Description:

Returns layout attribute of the graph.

Return:

Returns layout attribute of the graph.

get_negative_cycle(*self*)

API:

`get_negative_cycle(self)`

Description:

Finds and returns negative cost cycle using 'cost' attribute of arcs. Return value is a list of nodes representing cycle it is in the following form; n₁-n₂-...-n_k, when the cycle has k nodes.

Pre:

Arcs should have 'cost' attribute.

Return:

Returns a list of nodes in the cycle if a negative cycle exists, returns None otherwise.

get_neighbors(*self*, *name*)API: `get_neighbors(self, name)`

Description:

Returns list of neighbors of given node.

Input:

`name`: Node name.

Pre:

Graph should have this node.

Return:

List of neighbor node names.

get_node(*self*, *name*)

API: get_node(*self*, *name*)

Description:

Returns node object with the provided name.

Input:

name: Name of the node.

Return:

Returns node object if node exists, returns None otherwise.

get_node_attr(*self*, *name*, *attr*)

API: get_node_attr(*self*, *name*, *attr*)

Description:

Returns attribute attr of given node.

Input:

name: Name of node.

attr: Attribute of node.

Pre:

Graph should have this node.

Return:

Value of node attribute attr.

get_node_list(*self*)

API: get_node_list(*self*)

Description:

Returns node list.

Return:

List of nodes.

get_node_num(*self*)

API: get_node_num(*self*)

Description:

Returns number of nodes.

Return:

Number of nodes.

get_out_neighbors(*self*, *name*)

API: get_out_neighbors(*self*, *name*)

Description:

Returns list of out-neighbors of given node.

Input:

name: Node name.

Pre:

Graph should have this node.

Return:

List of out-neighbor node names.

get_simplex_solution_graph(*self*)

API:

get_simplex_solution_graph(*self*):

Description:

Assumes a feasible flow solution stored in 'flow' attribute's of arcs. Returns the graph with arcs that have flow between 0 and capacity.

Pre:

(1) 'flow' attribute represents a feasible flow solution. See Pre section of min_cost_flow() for details.

Return:

Graph instance that only has the arcs that have flow strictly between 0 and capacity.

label_components(*self*, *display*=None)

API: label_components(*self*, *display*=None)

Description:

This method labels the nodes of an undirected graph with component numbers so that each node has the same label as all nodes in the same component. It will display the algorithm if *display* argument is provided.

Input:

display: display method.

Pre:

self.graph.type should be UNDIRECTED_GRAPH.

Post:

Nodes will have 'component' attribute that will have component number as value.

label_correcting_check_cycle(*self*, *j*, *pred*)

API:

label_correcting_check_cycle(*self*, *j*, *pred*)

Description:

Checks if predecessor dictionary has a cycle, *j* represents the node that predecessor is recently updated.

Pre:

(1) predecessor of source node should be None.

Input:

j: node that predecessor is recently updated.

pred: predecessor dictionary

Return:

If there exists a cycle, returns the list that represents the cycle, otherwise it returns to None.

label_correcting_get_cycle(*self*, *j*, *pred*)

API:

```
label_correcting_get_cycle(self, labelled, pred)
```

Description:

In label correcting check cycle it is decided pred has a cycle and nodes in the cycle are labelled. We will create a list of nodes in the cycle using labelled and pred inputs.

Pre:

This method should be called from label_correcting_check_cycle(), unless you are sure about what you are doing.

Input:

j: Node that predecessor is recently updated. We know that it is in the cycle

pred: Predecessor dictionary that contains a cycle

Post:

Returns a list of nodes that represents cycle. It is in [*n*₁, *n*₂, ..., *n*_k] form where the cycle has *k* nodes.

label_strong_component(*self*)

API: label_strong_component(*self*)

Description:

This method labels the nodes of a directed graph with component numbers so that each node has the same label as all nodes in the same component.

Pre:

self.graph_type should be DIRECTED_GRAPH.

Post:

Nodes will have 'component' attribute that will have component number as value. Changes 'index' attribute of nodes.

max_flow(*self*, *source*, *sink*, *display=None*)API: `max_flow(self, source, sink, display=None)`

Description:

Finds maximum flow from source to sink by a depth-first search based augmenting path algorithm.

Pre:

Assumes a directed graph in which each arc has a 'capacity' attribute and for which there does not exist both arcs (i,j) and (j, i) for any pair of nodes i and j.

Input:

source: Source node name.

sink: Sink node name.

display: Display mode.

Post:

The 'flow' attribute of each arc gives a maximum flow.

max_flow_preflowpush(*self*, *source*, *sink*, *algo='FIFO'*, *display=None*)API: `max_flow_preflowpush(self, source, sink, algo = 'FIFO',
display = None)`

Description:

Finds maximum flow from source to sink by a depth-first search based augmenting path algorithm.

Pre:

Assumes a directed graph in which each arc has a 'capacity' attribute and for which there does not exist both arcs (i,j) and (j,i) for any pair of nodes i and j.

Input:

source: Source node name.

sink: Sink node name.

algo: Algorithm choice, 'FIFO', 'SAP' or 'HighestLabel'.

display: display method.

Post:

The 'flow' attribute of each arc gives a maximum flow.

```
min_cost_flow(self, display=None, **args)
```

API:

```
min_cost_flow(self, display='off', **args)
```

Description:

Solves minimum cost flow problem using node/edge attributes with the algorithm specified.

Pre:

- (1) Assumes a directed graph in which each arc has 'capacity' and 'cost' attributes.
- (2) Nodes should have 'demand' attribute. This value should be positive for supply and negative for demand, and 0 for transshipment nodes.
- (3) The graph should be connected.
- (4) Assumes (i,j) and (j,i) does not exist together. Needed when solving max flow. (max flow problem is solved to get a feasible flow).

Input:

display: 'off' for no display, 'pygame' for live update of tree

args: may have the following

display: display method, if not given current mode (the one specified by `__init__` or `set_display`) will be used.

algo: determines algorithm to use, can be one of the following

'simplex': network simplex algorithm

'cycle_canceling': cycle canceling algorithm

'simplex' is used if not given.

see Network Flows by Ahuja et al. for details of algorithms.

pivot: valid if algo is 'simplex', determines pivoting rule for

simplex, may be one of the following; 'first_eligible',

'dantzig' or 'scaled'.

'dantzig' is used if not given.

see Network Flows by Ahuja et al. for pivot rules.

root: valid if algo is 'simplex', specifies the root node for

simplex algorithm. It is name of the one of the nodes. It

will be chosen randomly if not provided.

Post:

The 'flow' attribute of each arc gives the optimal flows.

'distance' attribute of the nodes are also changed during max flow solution process.

Examples:

```
g.min_cost_flow():
```

solves minimum cost feasible flow problem using simplex algorithm with dantzig pivoting rule.

See pre section for details.

```
g.min_cost_flow(algo='cycle_canceling'):
```

solves minimum cost feasible flow problem using cycle canceling algorithm.

```
g.min_cost_flow(algo='simplex', pivot='scaled'):
```

solves minimum cost feasible flow problem using network simplex

minimum_spanning_tree_kruskal(*self*, *display*=None, *components*=None)

API: `minimum_spanning_tree_kruskal(self, display = None,
 components = None)`

Description:

Determines a minimum spanning tree using Kruskal's Algorithm.

Input:

display: Display method.

component: component number.

Post:

'color' attribute of nodes and edges may change.

Return:

Returns list of edges where edges are tuples in (source,sink) format.

minimum_spanning_tree_prim(*self*, *source*, *display*=None, *q*=PriorityQueue())

API: `minimum_spanning_tree_prim(self, source, display = None,
 q = PriorityQueue())`

Description:

Determines a minimum spanning tree of all nodes reachable from source using Prim's Algorithm.

Input:

source: Name of source node.

display: Display method.

q: Data structure that holds nodes to be processed in a queue.

Post:

'color', 'distance', 'component' attribute of nodes and edges may change.

Return:

Returns predecessor tree in dictionary format.

network_simplex(*self*, *display*, *pivot*, *root*)

API:

`network_simplex(self, display, pivot, root)`

Description:

Solves minimum cost feasible flow problem using network simplex algorithm. It is recommended to use `min_cost_flow(algo='simplex')` instead of using `network_simplex()` directly. Returns True when an optimal solution is found, returns False otherwise. 'flow' attribute values of arcs should be considered as junk when returned False.

Pre:

(1) check Pre section of `min_cost_flow()`

Input:

`pivot`: specifies pivot rule. Check `min_cost_flow()`
`display`: 'off' for no display, 'pygame' for live update of spanning tree.
`root`: Root node for the underlying spanning trees that will be generated by network simplex algorithm.

Post:

(1) Changes 'flow' attribute of edges.

Return:

Returns True when an optimal solution is found, returns False otherwise.

```
page_rank(self, damping_factor=0.85, max_iterations=100,  
min_delta=1e-05)
```

API:

```
page_rank(self, damping_factor=0.85, max_iterations=100,  
min_delta=0.00001)
```

Description:

Compute and return the page-rank of a directed graph.

This function was originally taken from here and modified for this graph class: <http://code.google.com/p/python-graph/source/browse/trunk/core/pygraph/algorithms/pagerank.py>

Input:

damping_factor: Damping factor.

max_iterations: Maximum number of iterations.

min_delta: Smallest variation required to have a new iteration.

Pre:

Graph should be a directed graph.

Return:

Returns dictionary of page-ranks. Keys are node names, values are corresponding page-ranks.

```
print_flow(self)
```

API:

```
print_flow(self)
```

Description:

Prints all positive flows to stdout. This method can be used for debugging purposes.

process_edge_dijkstra(*self, current, neighbor, pred, q, component*)

API: process_edge_dijkstra(*self, current, neighbor, pred, q, component*)

Description:

Used by search() method if the algo argument is 'Dijkstra'. Processes edges along Dijkstra's algorithm. User does not need to call this method directly.

Input:

current: Name of the current node.

neighbor: Name of the neighbor node.

pred: Predecessor tree.

q: Data structure that holds nodes to be processed in a queue.

component: component number.

Post:

'color' attribute of nodes and edges may change.

process_edge_flow(*self, source, sink, i, j, algo, q*)

API: process_edge_flow(*self, source, sink, i, j, algo, q*)

Description:

Used by by max_flow_preflowpush() method. Processes edges along preflow push.

Input:

source: Source node name of flow graph.

sink: Sink node name of flow graph.

i: Source node in the processed edge (tail of arc).

j: Sink node in the processed edge (head of arc).

Post:

The 'flow' and 'excess' attributes of nodes may get updated.

Return:

Returns False if residual capacity is 0, True otherwise.

`process_edge_prim(self, current, neighbor, pred, q, component)`

API: `process_edge_prim(self, current, neighbor, pred, q, component)`

Description:

Used by `search()` method if the `algo` argument is 'Prim'. Processes edges along Prim's algorithm. User does not need to call this method directly.

Input:

`current`: Name of the current node.

`neighbor`: Name of the neighbor node.

`pred`: Predecessor tree.

`q`: Data structure that holds nodes to be processed in a queue.

`component`: component number.

Post:

'color' attribute of nodes and edges may change.

`process_edge_search(self, current, neighbor, pred, q, component, algo, **kwargs)`

API: `process_edge_search(self, current, neighbor, pred, q, component, algo, **kwargs)`

Description:

Used by `search()` method. Processes edges according to the underlying algorithm. User does not need to call this method directly.

Input:

`current`: Name of the current node.

`neighbor`: Name of the neighbor node.

`pred`: Predecessor tree.

`q`: Data structure that holds nodes to be processed in a queue.

`component`: component number.

`algo`: Search algorithm. See `search()` documentation.

`kwargs`: Keyword arguments.

Post:

'color', 'distance', 'component' attribute of nodes and edges may change.

```
process_node_search(self, node, q, **kwargs)
```

API: `process_node_search(self, node, q, **kwargs)`

Description:

Used by `search()` method. Process nodes along the search. Should not be called by user directly.

Input:

node: Name of the node being processed.
 q: Queue data structure.
 kwargs: Keyword arguments.

Post:

'priority' attribute of the node may get updated.

```
random(self, numnodes=10, degree_range=None, length_range=None,
density=None, edge_format=None, node_format=None, Euclidean=False,
seedInput=0, add_labels=True, parallel_allowed=False)
```

API:

```
random(self, numnodes = 10, degree_range = None, length_range = None,
       density = None, edge_format = None, node_format = None,
       Euclidean = False, seedInput = 0)
```

Description:

Populates graph with random edges and nodes.

Input:

numnodes: Number of nodes to add.
 degree_range: A tuple that has lower and upper bounds of degree for a node.
 length_range: A tuple that has lower and upper bounds for 'cost' attribute of edges.
 density: Density of edges, ie. 0.5 indicates a node will approximately have edge to half of the other nodes.
 edge_format: Dictionary that specifies attribute values for edges.
 node_format: Dictionary that specifies attribute values for nodes.
 Euclidean: Creates an Euclidean graph (Euclidean distance between nodes) if True.
 seedInput: Seed that will be used for random number generation.

Pre:

It is recommended to call this method on empty Graph objects.

Post:

Graph will be populated by nodes and edges.

relabel(*self*, *i*)API: `relabel(self, i)`

Description:

Used by `max_flow_preflowpush()` method for relabelling node *i*.

Input:

i: Node that is being relabelled.

Post:

'distance' attribute of node *i* is updated.

```
search(self, source, destination=None, display=None, component=None,
q=None, algo='DFS', reverse=False, **kargs)
```

API: `search(self, source, destination = None, display = None, component = None, q = Stack(), algo = 'DFS', reverse = False, **kargs)`

Description:

Generic search method. Changes behavior (dfs,bfs,dijkstra,prim) according to algo argument.

if destination is not specified:

This method determines all nodes reachable from "source" ie. creates precedence tree and returns it (dictionary).

if destination is given:

If there exists a path from "source" to "destination" it will return list of the nodes in this path. If there is no such path, it will return the precedence tree constructed from source (dictionary).

Optionally, it marks all nodes reachable from "source" with a component number. The variable "q" determines the order in which the nodes are searched.

Input:

source: Search starts from node with this name.

destination: Destination node name.

display: Display method.

algo: Algorithm that specifies search. Available algorithms are 'DFS', 'BFS', 'Dijkstra' and 'Prim'.

reverse: Search goes in reverse arc directions if True.

kargs: Additional keyword arguments.

Post:

Nodes will have 'component' attribute that will have component number as value (if component argument provided). Color attribute of nodes and edges may change.

Return:

Returns predecessor tree in dictionary form if destination is not specified, returns list of node names in the path from source to destination if destination is specified and there is a path. If there is no path returns predecessor tree in dictionary form. See description section.

set_display_mode(*self*, *value*)

API:

`set_display_mode(self, value)`

Description:

Sets display mode to value.

Input:

value: New display mode.

Post:

Display mode attribute of graph is updated.

set_edge_attr(*self*, *n*, *m*, *attr*, *value*)API: `set_edge_attr(self, n, m, attr, value)`

Description:

Sets attr attribute of edge (n,m) to value.

Input:

n: Source node name.

m: Sink node name.

attr: Attribute of edge to set.

value: New value of attribute.

Pre:

Graph should have this edge.

Post:

Edge attribute will be updated.

set_layout(*self*, *value*)

API:

`set_layout(self, value)`

Description:

Sets layout attribute of the graph to value.

Input:

value: New value of the layout.

set_node_attr(*self*, *name*, *attr*, *value*)API: set_node_attr(*self*, *name*, *attr*)

Description:

Sets *attr* attribute of node named *name* to *value*.

Input:

name: Name of node. *attr*: Attribute of node to set.

Pre:

Graph should have this node.

Post:

Node attribute will be updated.

show_flow(*self*)API: relabel(*self*, *i*)

Description:

Used by max_flow_preflowpush() method for display purposed.

Post:

'color' and 'label' attribute of edges/nodes are updated.

simplex_augment_cycle(*self*, *cycle*)

API:

 simplex_augment_cycle(*self*, *cycle*)

Description:

Augments along the cycle to break it.

Pre:

'flow', 'capacity' attributes on arcs.

Input:

cycle: list representing a cycle in the solution

Post:

'flow' attribute will be modified.

simplex_compute_potentials(*self*, *t*, *root*)

API:

```
simplex_compute_potentials(self, t, root)
```

Description:

Computes node potentials for a minimum cost flow problem and stores them as node attribute 'potential'. Based on pseudocode given in Network Flows by Ahuja et al.

Pre:

- (1) Assumes a directed graph in which each arc has a 'cost' attribute.
- (2) Uses 'thread' and 'pred' attributes of nodes.

Input:

t: Current spanning tree solution, its type is Graph.
root: root node of the tree.

Post:

Keeps the node potentials as 'potential' attribute.

simplex_connect(*self*, *solution_g*)

API:

```
simplex_connect(self, solution_g)
```

Description:

At this point we assume that the solution does not have a cycle. We check if all the nodes are connected, if not we add an arc to *solution_g* that does not create a cycle and return True. Otherwise we do nothing and return False.

Pre:

- (1) We assume there is no cycle in the solution.

Input:

solution_g: current spanning tree solution instance.

Post:

- (1) *solution_g* is updated. An arc that does not create a cycle is added.
- (2) 'component' attribute of nodes are changed.

Return:

Returns True if an arc is added, returns False otherwise.

simplex_determine_leaving_arc(*self*, *t*, *k*, *l*)

API:

`simplex_determine_leaving_arc(self, t, k, l)`

Description:

Determines and returns the leaving arc.

Input:

t: current spanning tree solution.*k*: tail of the entering arc.*l*: head of the entering arc.

Return:

Returns the tuple that represents leaving arc, capacity of the cycle and cycle.

simplex_find_cycle(*self*)

API:

`simplex_find_cycle(self)`

Description:

Returns a cycle (list of nodes) if the graph has one, returns None otherwise. Uses DFS. During DFS checks existence of arcs to lower depth regions. Note that direction of the arcs are not important.

Return:

Returns list of nodes that represents cycle. Returns None if the graph does not have any cycle.

simplex_find_tree(*self*)

API:

`simplex_find_tree(self)`

Description:

Assumes a feasible flow solution stored in 'flow' attribute's of arcs and converts this solution to a feasible spanning tree solution.

Pre:

(1) 'flow' attributes represents a feasible flow solution.

Post:

(1) 'flow' attributes may change when eliminating cycles.

Return:

Return a Graph instance that is a spanning tree solution.

simplex_identify_cycle(*self*, *t*, *k*, *l*)

API:

```
identify_cycle(self, t, k, l)
```

Description:

Identifies and returns to the pivot cycle, which is a list of nodes.

Pre:

(1) *t* is spanning tree solution, (*k*,*l*) is the entering arc.

Input:

t: current spanning tree solution

k: tail of the entering arc

l: head of the entering arc

Returns:

List of nodes in the cycle.

simplex_mark_entering_arc(*self*, *k*, *l*)

API:

```
simplex.mark_entering_arc(self, k, l)
```

Description:

Marks entering arc (*k*,*l*)

Input:

k: tail of the entering arc

l: head of the entering arc

Post:

(1) color attribute of the arc (*k*,*l*)

simplex_mark_leaving_arc(*self*, *p*, *q*)

API:

```
simplex.mark_leaving_arc(self, p, q)
```

Description:

Marks leaving arc.

Input:

p: tail of the leaving arc

q: head of the leaving arc

Post:

Changes color attribute of leaving arc.

simplex_mark_st_arcs(*self*, *t*)

API:

`simplex_mark_st_arcs(self, t)`

Description:

Marks spanning tree arcs.

Case 1, Blue: Arcs that are at lower bound and in tree.

Case 2, Red: Arcs that are at upper bound and in tree.

Case 3, Green: Arcs that are between bounds are green.

Case 4, Brown: Non-tree arcs at lower bound.

Case 5, Violet: Non-tree arcs at upper bound.

Input:

`t`: `t` is the current spanning tree

Post:

(1) color attribute of edges.

simplex_optimal(*self*, *t*)

API:

`simplex_optimal(self, t)`

Description:

Checks if the current solution is optimal, if yes returns True, False otherwise.

Pre:

'flow' attributes represents a solution.

Input:

`t`: Graph instance that represents spanning tree solution.

Return:

Returns True if the current solution is optimal (optimality conditions are satisfied), else returns False

simplex_redraw(*self*, *display*, *root*)

API:

```
simplex_redraw(self, display, root)
```

Description:

Returns a new graph instance that is same as self but adds nodes and arcs in a way that the resulting tree will be displayed properly.

Input:

display: display mode
root: root node in tree.

Return:

Returns a graph same as self.

simplex_remove_arc(*self*, *t*, *p*, *q*, *min_capacity*, *cycle*)

API:

```
simplex_remove_arc(self, p, q, min_capacity, cycle)
```

Description:

Removes arc (p,q), updates t, updates flows, where (k,l) is the entering arc.

Input:

t: tree solution to be updated.
p: tail of the leaving arc.
q: head of the leaving arc.
min_capacity: capacity of the cycle.
cycle: cycle obtained when entering arc considered.

Post:

(1) updates t.
(2) updates 'flow' attributes.

simplex_search(*self*, *source*, *component_nr*)

API:

```
simplex_search(self, source, component_nr)
```

Description:

Searches graph starting from source. Its difference from usual search is we can also go backwards along an arc. When the graph is a spanning tree it computes predecessor, thread and depth indexes and stores them as node attributes. These values should be considered as junk when the graph is not a spanning tree.

Input:

source: source node
component_nr: component number

Post:

(1) Sets the component number of all reachable nodes to component. Changes 'component' attribute of nodes.
(2) Sets 'pred', 'thread' and 'depth' attributes of nodes. These values are junk if the graph is not a tree.

Return:

Returns predecessor dictionary.

simplex_select_entering_arc(*self*, *t*, *pivot*)

API:

```
simplex_select_entering_arc(self, t, pivot)
```

Description:

Decides and returns entering arc using pivot rule.

Input:

t: current spanning tree solution
pivot: May be one of the following; 'first_eligible' or 'dantzig'.
'dantzig' is the default value.

Return:

Returns entering arc tuple (k,l)

strong_connect(*self*, *q*, *node*, *index*, *component*)

API: strong_connect (self, q, node, index, component)

Description:

Used by tarjan method. This method should not be called directly by user.

Input:

q: Node list.

node: Node that is being connected to nodes in q.

index: Index used by tarjan method.

component: Current component number.

Pre:

Should be called by tarjan and itself (recursive) only.

Post:

Nodes will have 'component' attribute that will have component number as value. Changes 'index' attribute of nodes.

Return:

Returns new index and component numbers.

tarjan(*self*)

API: tarjan(self)

Description:

Implements Tarjan's algorithm for determining strongly connected set of nodes.

Pre:

self.graph.type should be DIRECTED_GRAPH.

Post:

Nodes will have 'component' attribute that will have component number as value. Changes 'index' attribute of nodes.

to_string(*self*)

API: to_string(self)

Description:

This method is based on pydot Graph class with the same name.

Returns a string representation of the graph in dot language.

It will return the graph and all its subelements in string form.

Return:

String that represents graph in dot language.

```
write(self, basename='graph', layout=None, format='png')
```

API:
 write(self, basename = 'graph', layout = None, format='png')

Description:
 Writes graph to dist using layout and format.

Input:
 basename: name of the file that will be written.
 layout: Dot layout for generating graph image.
 format: Image format, all format supported by Dot are wellcome.

Post:
 File will be written to disk.

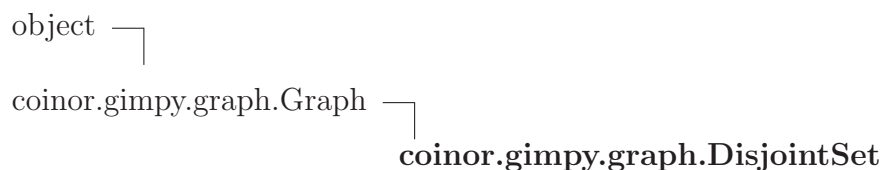
Inherited from object

```
--delattr--(), --format--(), --getattribute--(), --hash--(), --new--(), --reduce--(), --reduce_ex--(),  
--setattr--(), --sizeof--(), --str--(), --subclasshook--()
```

3.3.2 Properties

| Name | Description |
|------------------------------|-------------|
| <i>Inherited from object</i> | |
| <code>--class--</code> | |

3.4 Class DisjointSet



Disjoint set data structure. Inherits Graph class.

3.4.1 Methods

`__init__(self, optimize=True, **attrs)`

API:

`__init__(self, optimize = True, **attrs):`

Description:

Class constructor.

Input:

`optimize`: Optimizes `find()` if `True`.`attrs`: Graph attributes.

Post:

`self.optimize` will be updated.Overrides: `object.__init__`

`add(self, aList)`

API:

`add(self, aList)`

Description:

Adds items in the list to the set.

Input:

`aList`: List of items.

Post:

`self.sizes` will be updated.

`union(self, i, j)`

API:

`union(self, i, j):`

Description:

Finds sets of `i` and `j` and unites them.

Input:

`i`: Item.`j`: Item.

Post:

`self.sizes` will be updated.


```
find(self, i)
```

API:

```
find(self, i)
```

Description:

Returns root of set that has i.

Input:

i: Item.

Return:

Returns root of set that has i.

Inherited from coinor.gimpy.graph.Graph(Section 3.3)

`_repr_()`, `add_edge()`, `add_node()`, `augment_cycle()`, `bfs()`, `check_edge()`, `create()`, `create_cluster()`, `create_residual_graph()`, `cycle_canceling()`, `del_edge()`, `del_node()`, `dfs()`, `display()`, `edge_to_string()`, `fifo_label_correcting()`, `find_cycle_capacity()`, `find_feasible_flow()`, `get_approximate_diameter()`, `get_degree()`, `get_diameter()`, `get_edge_attr()`, `get_edge_cost()`, `get_edge_list()`, `get_edge_num()`, `get_in_neighbors()`, `get_layout()`, `get_negative_cycle()`, `get_neighbors()`, `get_node()`, `get_node_attr()`, `get_node_list()`, `get_node_num()`, `get_out_neighbors()`, `get_simplex_solution_graph()`, `label_components()`, `label_correcting_check_cycle()`, `label_correcting_get_cycle()`, `label_strong_component()`, `max_flow()`, `max_flow_preflowpush()`, `min_cost_flow()`, `minimum_spanning_tree_kruskal()`, `minimum_spanning_tree_prim()`, `network_simplex()`, `page_rank()`, `print_flow()`, `process_edge_dijkstra()`, `process_edge_flow()`, `process_edge_prim()`, `process_edge_search()`, `process_node_search()`, `random()`, `relabel()`, `search()`, `set_display_mode()`, `set_edge_attr()`, `set_layout()`, `set_node_attr()`, `show_flow()`, `simplex_augment_cycle()`, `simplex_compute_potentials()`, `simplex_connect()`, `simplex_determine_leaving_arc()`, `simplex_find_cycle()`, `simplex_find_tree()`, `simplex_identify_cycle()`, `simplex_mark_entering_arc()`, `simplex_mark_leaving_arc()`, `simplex_mark_st_arcs()`, `simplex_optimal()`, `simplex_redraw()`, `simplex_remove_arc()`, `simplex_search()`, `simplex_select_entering_arc()`, `strong_connect()`, `tarjan()`, `to_string()`, `write()`

Inherited from object

`__delattr__()`, `__format__()`, `__getattr__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__sizeof__()`, `__str__()`, `__subclasshook__()`

3.4.2 Properties

| Name | Description |
|------------------------------|-------------|
| <i>Inherited from object</i> | |
| <code>__class__</code> | |

4 Module `coinor.gimpy.tree`

Tree class built on top of Graph class.

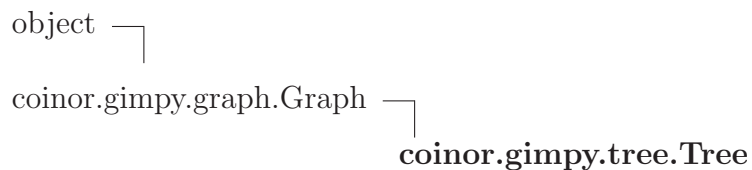
4.1 Variables

| Name | Description |
|-----------------------------|---|
| CLUSTER_ATTRIBUTES | Value: <code>set(['K', 'URL', 'bgcolor', 'color', 'colorscheme', 'fill...])</code> |
| DEFAULT_EDGE_ATTRIBUTES | Value: <code>{}</code> |
| DEFAULT_GRAPH_ATTRIBUTES | Value: <code>{}</code> |
| DEFAULT_NODE_ATTRIBUTES | Value: <code>{}</code> |
| DIRECTED_GRAPH | Value: <code>'digraph'</code> |
| DOT2TEX_INSTALLED | Value: <code>None</code> |
| DOT2TEX_TEMPLATE | Value: <code>'\n\\documentclass[landscape]{article}\n\\usepackage[x11n...]</code> |
| DOT_KEYWORDS | Value: <code>['graph', 'subgraph', 'digraph', 'node', 'edge', 'strict']</code> |
| EDGE_ATTRIBUTES | Value: <code>set(['URL', 'arrowhead', 'arrowsize', 'arrowtail', 'color...])</code> |
| EDGE_CONNECT_SYMBOL | Value: <code>{'digraph': ' -> ', 'graph': ' -- '}</code> |
| ETREE_INSTALLED | Value: <code>None</code> |
| GRAPH_ATTRIBUTES | Value: <code>set(['Damping', 'K', 'URL', 'aspect', 'bb', 'bgcolor', 'c...])</code> |
| ID_RE_ALPHA_NUMS | Value: <code>re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_]*\$')</code> |
| ID_RE_ALPHA_NUMS_WITH_PORTS | Value: <code>re.compile(r'(?u)^[a-zA-Z][a-zA-Z0-9_:"]*[a-zA-Z0-9_,"]..</code> |
| ID_RE_DBL_QUOTED | Value: <code>re.compile(r'(?su)^\".*\"\$')</code> |
| ID_RE_HTML | Value: <code>re.compile(r'(?su)^<.*>\$')</code> |
| ID_RE_NUM | Value: <code>re.compile(r'(?u)^[0-9,]+\$')</code> |
| ID_RE_WITH_PORT | Value: <code>re.compile(r'(?u)^([~:]*):([~:]*)\$')</code> |
| INF | Value: <code>10000</code> |
| NODE_ATTRIBUTES | Value: <code>set(['URL', 'color', 'colorscheme', 'comment', 'distortio...])</code> |
| PIL_INSTALLED | Value: <code>None</code> |
| PYGAME_INSTALLED | Value: <code>None</code> |
| UNDIRECTED_GRAPH | Value: <code>'graph'</code> |

continued on next page

| Name | Description |
|----------------|------------------------------|
| XDOT_INSTALLED | Value: None |
| --package-- | Value: 'coinor.gimpy' |

4.2 Class Tree



Known Subclasses: coinor.gimpy.tree.BinaryTree

Tree class. It inherits from Graph class. Provides DFS, BFS and traverse methods.

4.2.1 Methods

```
__init__(self, **attrs)
```

API: `__init__(self, **attrs)`

Description:

Constructor. Sets attributes of class using argument.

Input:

`attrs`: Attributes in keyword arguments format.

Overrides: `object.__init__`

```
get_children(self, n)
```

API: `get_children(self, n)`

Description:

Returns list of children of node `n`.

Pre:

Node with name `n` should exist.

Input:

`n`: Node name.

Return:

Returns list of names of children nodes of `n`.

get_parent(*self*, *n*)

API: get_parent(*self*, *n*)

Description:

Returns parent node name if *n*'s parent exists, returns None otherwise.

Pre:

Node with name *n* should exist.

Input:

n: Node name.

Return:

Returns parent name of *n* if its parent exists, returns None otherwise.

add_root(*self*, *root*, *******attrs*)

API: add_root(*self*, *root*, *******attrs*)

Description:

Adds root node to the tree with name *root* and returns root Node instance.

Input:

root: Root node name.

attrs: Root node attributes.

Post:

Changes *self.root*.

Return:

Returns root Node instance.

add_child(*self*, *n*, *parent*, ****attrs**)API: add_child(*self*, *n*, *parent*, ****attrs**)

Description:

Adds child *n* to node *parent* and return Node *n*.

Pre:

Node with name *parent* should exist.

Input:

n: Child node name.*parent*: Parent node name.*attrs*: Attributes of node being added.

Post:

Updates Graph related graph data attributes.

Return:

Returns *n* Node instance.

dfs(*self*, *root*=None, *display*=None)API: dfs(*self*, *root* = None, *display* = None)

Description:

Searches tree starting from node named *root* using depth-first strategy if *root* argument is provided. Starts search from *root* node of the tree otherwise.

Pre:

Node indicated by *root* argument should exist.

Input:

root: Starting node name.*display*: Display argument.

Overrides: coinor.gimpy.graph.Graph.dfs

bfs(*self*, *root*=None, *display*=None)

API: `bfs(self, root = None, display = None)`

Description:

Searches tree starting from node named *root* using breadth-first strategy if *root* argument is provided. Starts search from *root* node of the tree otherwise.

Pre:

Node indicated by *root* argument should exist.

Input:

root: Starting node name.

display: Display argument.

Overrides: `coinor.gimpy.graph.Graph.bfs`

traverse(*self*, *root*=None, *display*=None, *q*=[])

API: `traverse(self, root = None, display = None, q = Stack())`

Description:

Traverses tree starting from node named *root*. Used strategy (BFS, DFS) is controlled by argument *q*. It is a DFS if *q* is `Queue()`, BFS if *q* is `Stack()`. Starts search from *root* argument if it is given. Starts from *root* node of the tree otherwise.

Pre:

Node indicated by *root* argument should exist.

Input:

root: Starting node name.

display: Display argument.

q: Queue data structure instance. It is either a `Stack()` or `Queue()`.

Inherited from coinor.gimpy.graph.Graph(Section 3.3)

`__repr__()`, `add_edge()`, `add_node()`, `augment_cycle()`, `check_edge()`, `create()`, `create_cluster()`, `create_residual_graph()`, `cycle_canceling()`, `del_edge()`, `del_node()`, `display()`, `edge_to_string()`, `fifo_label_correcting()`, `find_cycle_capacity()`, `find_feasible_flow()`, `get_approximate_diameter()`, `get_degree()`, `get_diameter()`, `get_edge_attr()`, `get_edge_cost()`, `get_edge_list()`, `get_edge_num()`, `get_in_neighbors()`, `get_layout()`, `get_negative_cycle()`, `get_neighbors()`, `get_node()`, `get_node_attr()`, `get_node_list()`, `get_node_num()`, `get_out_neighbors()`, `get_simplex_solution_graph()`, `label_components()`, `label_correcting_check_cycle()`, `label_correcting_get_cycle()`, `label_strong_component()`, `max_flow()`, `max_flow_preflowpush()`, `min_cost_flow()`, `minimum_spanning_tree_kruskal()`, `minimum_spanning_tree_prim()`, `network_simplex()`, `page_rank()`, `print_flow()`, `process_edge_dijkstra()`, `process_edge_flow()`,

process_edge_prim(), process_edge_search(), process_node_search(), random(), re-label(), search(), set_display_mode(), set_edge_attr(), set_layout(), set_node_attr(), show_flow(), simplex_augment_cycle(), simplex_compute_potentials(), simplex_connect(), simplex_determine_leaving_arc(), simplex_find_cycle(), simplex_find_tree(), simplex_identify_cycle(), simplex_mark_entering_arc(), simplex_mark_leaving_arc(), simplex_mark_st_arcs(), simplex_optimal(), simplex_redraw(), simplex_remove_arc(), simplex_search(), simplex_select_entering_arc(), strong_connect(), tarjan(), to_string(), write()

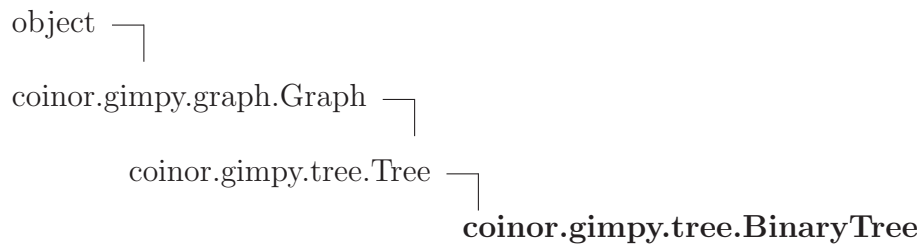
Inherited from object

__delattr__(), __format__(), __getattr__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.2.2 Properties

| Name | Description |
|------------------------------|-------------|
| <i>Inherited from object</i> | |
| __class__ | |

4.3 Class **BinaryTree**



Binary tree class. Inherits Tree class. Provides methods for adding left/right childs and binary tree specific DFS and BFS methods.

4.3.1 Methods

`__init__(self, **attrs)`

API: `__init__(self, **attrs)`

Description:

Class constructor.

Input:

`attrs`: Tree attributes in keyword arguments format. See Graph and Tree class for details.

Overrides: `object.__init__`

`add_root(self, root, **attrs)`

API: `add_root(self, root, **attrs)`

Description:

Adds root node to the binary tree.

Input:

`root`: Name of the root node.

`attrs`: Attributes of the root node.

Post:

Changes `self.root` attribute.

Overrides: `coinor.gimpy.tree.Tree.add_root`

`add_right_child(self, n, parent, **attrs)`

API: `add_right_child(self, n, parent, **attrs)`

Description:

Adds right child `n` to node `parent`.

Pre:

Right child of `parent` should not exist.

Input:

`n`: Node name.

`parent`: Parent node name.

`attrs`: Attributes of node `n`.

add_left_child(*self*, *n*, *parent*, ****attrs**)API: `add_left_child(self, n, parent, **attrs)`

Description:

Adds left child *n* to node *parent*.

Pre:

Left child of *parent* should not exist.

Input:

n: Node name.*parent*: Parent node name.*attrs*: Attributes of node *n*.

get_right_child(*self*, *n*)API: `get_right_child(self, n)`

Description:

Returns right child of node *n*. *n* can be `Node()` instance or string (name of node).

Pre:

Node *n* should be present in the tree.

Input:

n: Node name or `Node()` instance.

Return:

Returns name of the right child of *n*.

get_left_child(*self*, *n*)API: `get_left_child(self, n)`

Description:

Returns left child of node *n*. *n* can be `Node()` instance or string (name of node).

Pre:

Node *n* should be present in the tree.

Input:

n: Node name or `Node()` instance.

Return:

Returns name of the left child of *n*.

del_node(*self*, *n*)

API: `del_node(self, n)`

Description:

Removes node *n* from tree.

Pre:

Node *n* should be present in the tree.

Input:

n: Node name.Overrides: `coinor.gimpy.graph.Graph.del_node`

print_nodes(*self*, *order*='in', *priority*='L', *display*=None, *root*=None)

API: `print_nodes(self, order = 'in', priority = 'L', display = None, root = None)`

Description:

A recursive function that prints nodes to stdout starting from *root*.

Input:

order: Order of printing. Acceptable arguments are 'pre', 'in', 'post'.*priority*: Priority of printing, acceptable arguments are 'L' and 'R'.*display*: Display mode.*root*: Starting node.

dfs(*self*, *root*=None, *display*=None, *priority*='L')

API: `dfs(self, root=None, display=None, priority='L', order='in')`

Description:

Searches tree starting from node named *root* using depth-first strategy if *root* argument is provided. Starts search from *root* node of the tree otherwise.

Input:

root: Starting node.*display*: Display mode.*priority*: Priority used when exploring children of the node. Acceptable arguments are 'L' and 'R'.Overrides: `coinor.gimpy.graph.Graph.dfs`

```
bfs(self, root=None, display=None, priority='L')
```

API: bfs(self, root=None, display=None, priority='L', order='in')

Description:

Searches tree starting from node named root using breadth-first strategy if root argument is provided. Starts search from root node of the tree otherwise.

Input:

root: Starting node.
display: Display mode.
priority: Priority used when exploring children of the node.
Acceptable arguments are 'L' and 'R'.

Overrides: coinor.gimpy.graph.Graph.bfs

```
traverse(self, root=None, display=None, q=[], priority='L')
```

API: traverse(self, root=None, display=None, q=Stack(), priority='L', order='in')

Description:

Traverses tree starting from node named root if root argument is provided. Starts search from root node of the tree otherwise. Search strategy is determined by q data structure. It is DFS if q is Stack() and BFS if Queue().

Input:

root: Starting node.
display: Display mode.
q: Queue data structure, either Queue() or Stack().
priority: Priority used when exploring children of the node.
Acceptable arguments are 'L' and 'R'.
order: Ineffective, will be removed.

Overrides: coinor.gimpy.tree.Tree.traverse

```
printexp(self, display=None, root=None)
```

```
postordereval(self, display=None, root=None)
```

Inherited from coinor.gimpy.tree.Tree(Section 4.2)

add_child(), get_children(), get_parent()

Inherited from coinor.gimpy.graph.Graph(Section 3.3)

`__repr__()`, `add_edge()`, `add_node()`, `augment_cycle()`, `check_edge()`, `create()`, `create_cluster()`, `create_residual_graph()`, `cycle_canceling()`, `del_edge()`, `display()`, `edge_to_string()`, `fifo_label_correcting()`, `find_cycle_capacity()`, `find_feasible_flow()`, `get_approximate_diameter()`, `get_degree()`, `get_diameter()`, `get_edge_attr()`, `get_edge_cost()`, `get_edge_list()`, `get_edge_num()`, `get_in_neighbors()`, `get_layout()`, `get_negative_cycle()`, `get_neighbors()`, `get_node()`, `get_node_attr()`, `get_node_list()`, `get_node_num()`, `get_out_neighbors()`, `get_simplex_solution_graph()`, `label_components()`, `label_correcting_check_cycle()`, `label_correcting_get_cycle()`, `label_strong_component()`, `max_flow()`, `max_flow_preflowpush()`, `min_cost_flow()`, `minimum_spanning_tree_kruskal()`, `minimum_spanning_tree_prim()`, `network_simplex()`, `page_rank()`, `print_flow()`, `process_edge_dijkstra()`, `process_edge_flow()`, `process_edge_prim()`, `process_edge_search()`, `process_node_search()`, `random()`, `relabel()`, `search()`, `set_display_mode()`, `set_edge_attr()`, `set_layout()`, `set_node_attr()`, `show_flow()`, `simplex_augment_cycle()`, `simplex_compute_potentials()`, `simplex_connect()`, `simplex_determine_leaving_arc()`, `simplex_find_cycle()`, `simplex_find_tree()`, `simplex_identify_cycle()`, `simplex_mark_entering_arc()`, `simplex_mark_leaving_arc()`, `simplex_mark_st_arcs()`, `simplex_optimal()`, `simplex_redraw()`, `simplex_remove_arc()`, `simplex_search()`, `simplex_select_entering_arc()`, `strong_connect()`, `tarjan()`, `to_string()`, `write()`

Inherited from object

`__delattr__()`, `__format__()`, `__getattribute__()`, `__hash__()`, `__new__()`, `__reduce__()`, `__reduce_ex__()`, `__setattr__()`, `__sizeof__()`, `__str__()`, `__subclasshook__()`

4.3.2 Properties

| Name | Description |
|------------------------------|-------------|
| <i>Inherited from object</i> | |
| <code>__class__</code> | |

Index

coinor (*package*)

 coinor.gimp (*package*), 2–3

 coinor.gimp.global_constants (*module*),
 4–6

 coinor.gimp.graph (*module*), 7–49

 coinor.gimp.tree (*module*), 50–60