Sage 9.1 Reference Manual: Graph Theory

Release 9.1

The Sage Development Team

May 21, 2020

CONTENTS

1 Graph objects and methods 1

2 Constructors and databases 403

3 Low-level implementation 601

4 Hypergraphs 681

5 Libraries of algorithms 701

6 Indices and Tables 961

Bibliography 963

Python Module Index 965

Index 967

CHAPTER

ONE

GRAPH OBJECTS AND METHODS

1.1 Generic graphs (common to directed/undirected)

This module implements the base class for graphs and digraphs, and methods that can be applied on both. Here is what it can do:

Basic Graph operations:

networkx_graph()	Return a new NetworkX graph from the Sage graph
igraph_graph()	Return an igraph graph from the Sage graph
to_dictionary()	Create a dictionary encoding the graph.
copy()	Return a copy of the graph.
export_to_file()	Export the graph to a file.
adjacency_matrix()	Return the adjacency matrix of the (di)graph.
incidence_matrix()	Return an incidence matrix of the (di)graph
distance_matrix()	Return the distance matrix of the (strongly) connected (di)graph
weighted_adjacency_matrix()	Return the weighted adjacency matrix of the graph
kirchhoff_matrix()	Return the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.
has_loops()	Return whether there are loops in the (di)graph
allows_loops()	Return whether loops are permitted in the (di)graph
allow_loops()	Change whether loops are permitted in the (di)graph
loops()	Return a list of all loops in the (di)graph
loop_edges()	Return a list of all loops in the (di)graph
number_of_loops()	Return the number of edges that are loops
loop_vertices()	Return a list of vertices with loops
remove_loops()	Remove loops on vertices in vertices.
has_multiple_edges()	Return whether there are multiple edges in the (di)graph.
allows_multiple_edges()	Return whether multiple edges are permitted in the (di)graph.
allow_multiple_edges()	Change whether multiple edges are permitted in the (di)graph.
multiple_edges()	Return any multiple edges in the (di)graph.
name()	Return or set the graph's name.
is_immutable()	Return whether the graph is immutable.
weighted()	Whether the (di)graph is to be considered as a weighted (di)graph.
antisymmetric()	Test whether the graph is antisymmetric
density()	Return the density
order()	Return the number of vertices.
size()	Return the number of edges.
add_vertex()	Create an isolated vertex.
add_vertices()	Add vertices to the (di)graph from an iterable container of vertices

Continued on next page

Table 1 – continued from previous page

delete_vertex()	Delete vertex, removing all incident edges.
delete_vertices()	Delete vertices from the (di)graph taken from an iterable container of vertices.
has_vertex()	Check if `vertex` is one of the vertices of this graph.
random_vertex()	Return a random vertex of `self`.
random_vertex_iterator()	Return an iterator over random vertices of `self`.
random_edge()	Return a random edge of `self`.
random_edge_iterator()	Return an iterator over random edges of `self`.
vertex_boundary()	Return a list of all vertices in the external boundary of `vertices1`, intersected with `vertices2`.
set_vertices()	Associate arbitrary objects with each vertex
set_vertex()	Associate an arbitrary object with a vertex.
get_vertex()	Retrieve the object associated with a given vertex.
get_vertices()	Return a dictionary of the objects associated to each vertex.
vertex_iterator()	Return an iterator over the given vertices.
neighbor_iterator()	Return an iterator over neighbors of `vertex`.
vertices()	Return a list of the vertices.
neighbors()	Return a list of neighbors (in and out if directed) of `vertex`.
merge_vertices()	Merge vertices.
add_edge()	Add an edge from `u` to `v`.
add_edges()	Add edges from an iterable container.
subdivide_edge()	Subdivide an edge k times.
subdivide_edges()	Subdivide k times edges from an iterable container.
delete_edge()	Delete the edge from `u` to `v`
delete_edges()	Delete edges from an iterable container.
contract_edge()	Contract an edge from `u` to `v`.
contract_edges()	Contract edges from an iterable container.
delete_multiedge()	Delete all edges from `u` to `v`.
set_edge_label()	Set the edge label of a given edge.
has_edge()	Check whether `(u, v)` is an edge of the (di)graph.
edges()	Return a *EdgesView* of edges.
edge_boundary()	Return a list of edges `(u,v,l)` with `u` in `vertices1`
edge_iterator()	Return an iterator over edges.
edges_incident()	Return incident edges to some vertices.
edge_label()	Return the label of an edge.
edge_labels()	Return a list of the labels of all edges in `self`.
remove_multiple_edges()	Remove all multiple edges, retaining one edge for each.
clear()	Empty the graph of vertices and edges and removes name, associated objects, and position information.
degree()	Return the degree (in + out for digraphs) of a vertex or of vertices.
average_degree()	Return the average degree of the graph.
degree_histogram()	Return a list, whose ith entry is the frequency of degree i.
degree_iterator()	Return an iterator over the degrees of the (di)graph.
degree_sequence()	Return the degree sequence of this (di)graph.
random_subgraph()	Return a random subgraph containing each vertex with probability `p`.
add_clique()	Add a clique to the graph with the given vertices.
add_cycle()	Add a cycle to the graph with the given vertices.
add_path()	Add a path to the graph with the given vertices.
complement()	Return the complement of the (di)graph.
line_graph()	Return the line graph of the (di)graph.

Continued on next page

Table 1 – continued from previous page

to_simple()	Return a simple version of itself (i.e., undirected and loops and multiple edges are removed).
disjoint_union()	Return the disjoint union of self and other.
union()	Return the union of self and other.
relabel()	Relabel the vertices of `self`
degree_to_cell()	Return the number of edges from vertex to an edge in cell.
subgraph()	Return the subgraph containing the given vertices and edges.
is_subgraph()	Check whether `self` is a subgraph of `other`.

Graph products:

cartesian_product()	Return the Cartesian product of self and other.
tensor_product()	Return the tensor product, also called the categorical product, of self and other.
lexicographic_product()	Return the lexicographic product of self and other.
strong_product()	Return the strong product of self and other.
disjunctive_product()	Return the disjunctive product of self and other.

Paths and cycles:

eulerian_orientation()	Return a DiGraph which is an Eulerian orientation of the current graph.
eulerian_circuit()	Return a list of edges forming an Eulerian circuit if one exists.
minimum_cycle_basis()	Return a minimum weight cycle basis of the graph.
cycle_basis()	Return a list of cycles which form a basis of the cycle space of `self`.
all_paths()	Return a list of all paths (also lists) between a pair of vertices in the (di)graph.
triangles_count()	Return the number of triangles in the (di)graph.
shortest_simple_paths()	Return an iterator over the simple paths between a pair of vertices.

Linear algebra:

spectrum()	Return a list of the eigenvalues of the adjacency matrix.
eigenvectors()	Return the *right* eigenvectors of the adjacency matrix of the graph.
eigenspaces()	Return the *right* eigenspaces of the adjacency matrix of the graph.

Some metrics:

cluster_triangles()	Return the number of triangles for the set nbunch of vertices as a dictionary keyed by vertex.
clustering_average()	Return the average clustering coefficient.
clustering_coeff()	Return the clustering coefficient for each vertex in nbunch
cluster_transitivity()	Return the transitivity (fraction of transitive triangles) of the graph.
szeged_index()	Return the Szeged index of the graph.
katz_centrality()	Return the katz centrality of the vertex u of the graph.
katz_matrix()	Return the katz matrix of the graph.
pagerank()	Return the PageRank of the vertices of `self`.

Automorphism group:

coarsest_equitable_refinement()	Return the coarsest partition which is finer than the input partition, and equitable with respect to self.
automorphism_group()	Return the largest subgroup of the automorphism group of the (di)graph whose orbit partition is finer than the partition given.
is_vertex_transitive()	Return whether the automorphism group of self is transitive within the partition provided
is_isomorphic()	Test for isomorphism between self and other.
canonical_label()	Return the canonical graph.
is_cayley()	Check whether the graph is a Cayley graph.

Graph properties:

is_eulerian()	Return `True` if the graph has a (closed) tour that visits each edge exactly once.
is_planar()	Check whether the graph is planar.
is_circular_planar()	Check whether the graph is circular planar (outerplanar)
is_regular()	Return `True` if this graph is (k-)regular.
is_chordal()	Check whether the given graph is chordal.
is_bipartite()	Test whether the given graph is bipartite.
is_circulant()	Check whether the graph is a circulant graph.
is_interval()	Check whether the graph is an interval graph.
is_gallai_tree()	Return whether the current graph is a Gallai tree.
is_clique()	Check whether a set of vertices is a clique
is_cycle()	Check whether `self` is a (directed) cycle graph.
is_independent_set()	Check whether `vertices` is an independent set of `self`
is_transitively_reduced()	Test whether the digraph is transitively reduced.
is_equitable()	Check whether the given partition is equitable with respect to self.
is_self_complementary()	Check whether the graph is self-complementary.

Traversals:

breadth_first_search()	Return an iterator over the vertices in a breadth-first ordering.
depth_first_search()	Return an iterator over the vertices in a depth-first ordering.
lex_BFS()	Perform a lexicographic breadth first search (LexBFS) on the graph.
lex_UP()	Perform a lexicographic UP search (LexUP) on the graph.
lex_DFS()	Perform a lexicographic depth first search (LexDFS) on the graph.
lex_DOWN()	Perform a lexicographic DOWN search (LexDOWN) on the graph.

Distances:

centrality_betweenness()	Return the betweenness centrality
centrality_closeness()	Returns the closeness centrality (1/average distance to all vertices)
distance()	Return the (directed) distance from u to v in the (di)graph
distance_all_pairs()	Return the distances between all pairs of vertices.
distances_distribution()	Return the distances distribution of the (di)graph in a dictionary.
eccentricity()	Return the eccentricity of vertex (or vertices) v.
radius()	Return the radius of the (di)graph.
center()	Return the set of vertices in the center of the graph
diameter()	Return the largest distance between any two vertices.
distance_graph()	Return the graph on the same vertex set as the original graph but vertices are adjacent in the returned graph if and only if they are at specified distances in the original graph.
girth()	Return the girth of the graph.
odd_girth()	Return the odd girth of the graph.
periphery()	Return the set of vertices in the periphery
shortest_path()	Return a list of vertices representing some shortest path from u to v
shortest_path_length()	Return the minimal length of paths from u to v
shortest_paths()	Return a dictionary associating to each vertex v a shortest path from u to v, if it exists.
shortest_path_lengths()	Return a dictionary of shortest path lengths keyed by targets that are connected by a path from u.
shortest_path_all_pairs()	Compute a shortest path between each pair of vertices.
wiener_index()	Return the Wiener index of the graph.
average_distance()	Return the average distance between vertices of the graph.

Flows, connectivity, trees:

is_connected()	Test whether the (di)graph is connected.
connected_components()	Return the list of connected components
connected_components_number()	Return the number of connected components.
connected_components_subgraphs()	Return a list of connected components as graph objects.
connected_component_containing_vertex()	Return a list of the vertices connected to vertex.
connected_components_sizes()	Return the sizes of the connected components as a list.
blocks_and_cut_vertices()	Compute the blocks and cut vertices of the graph.
blocks_and_cuts_tree()	Compute the blocks-and-cuts tree of the graph.
is_cut_edge()	Return True if the input edge is a cut-edge or a bridge.
is_cut_vertex()	Return True if the input vertex is a cut-vertex.
edge_cut()	Return a minimum edge cut between vertices s and t
vertex_cut()	Return a minimum vertex cut between non-adjacent vertices s and t
flow()	Return a maximum flow in the graph from `x` to `y`
nowhere_zero_flow()	Return a k-nowhere zero flow of the (di)graph.
edge_disjoint_paths()	Return a list of edge-disjoint paths between two vertices
vertex_disjoint_paths()	Return a list of vertex-disjoint paths between two vertices
edge_connectivity()	Return the edge connectivity of the graph.
vertex_connectivity()	Return the vertex connectivity of the graph.
transitive_closure()	Compute the transitive closure of a graph and returns it.
transitive_reduction()	Return a transitive reduction of a graph.
min_spanning_tree()	Return the edges of a minimum spanning tree.
spanning_trees_count()	Return the number of spanning trees in a graph.
dominator_tree()	Returns a dominator tree of the graph.
connected_subgraph_iterator()	Iterator over the induced connected subgraphs of order at most k

Plot/embedding-related methods:

set_embedding()	Set a combinatorial embedding dictionary to _embedding attribute.
get_embedding()	Return the attribute _embedding if it exists.
faces()	Return the faces of an embedded graph.
genus()	Return the number of faces of an embedded graph.
planar_dual()	Return the planar dual of an embedded graph.
get_pos()	Return the position dictionary
set_pos()	Set the position dictionary.
set_planar_positions()	Compute a planar layout for self using Schnyder's algorithm
layout_planar()	Compute a planar layout of the graph using Schnyder's algorithm.
is_drawn_free_of_edge_crossings()	Check whether the position dictionary gives a planar embedding.
latex_options()	Return an instance of *GraphLatex* for the graph.
set_latex_options()	Set multiple options for rendering a graph with LaTeX.
layout()	Return a layout for the vertices of this graph.
layout_spring()	Return a spring layout for this graph
layout_ranked()	Return a ranked layout for this graph
layout_extend_randomly()	Extend randomly a partial layout
layout_circular()	Return a circular layout for this graph
layout_tree()	Return an ordered tree layout for this graph
layout_graphviz()	Call graphviz to compute a layout of the vertices of this graph.
_circle_embedding()	Set some vertices on a circle in the embedding of this graph.
_line_embedding()	Set some vertices on a line in the embedding of this graph.
graphplot()	Return a *GraphPlot* object.
plot()	Return a Graphics object representing the (di)graph.
show()	Show the (di)graph.
plot3d()	Plot the graph in three dimensions.
show3d()	Plot the graph using Tachyon, and shows the resulting plot.
graphviz_string()	Return a representation in the dot language.
graphviz_to_file_named()	Write a representation in the dot language in a file.

Algorithmically hard stuff:

steiner_tree()	Return a tree of minimum weight connecting the given set of vertices.
edge_disjoint_spanning_trees()	Return the desired number of edge-disjoint spanning trees/arborescences.
feedback_vertex_set()	Compute the minimum feedback vertex set of a (di)graph.
multiway_cut()	Return a minimum edge multiway cut
max_cut()	Return a maximum edge cut of the graph.
longest_path()	Return a longest path of self.
traveling_salesman_problem()	Solve the traveling salesman problem (TSP)
is_hamiltonian()	Test whether the current graph is Hamiltonian.
hamiltonian_cycle()	Return a Hamiltonian cycle/circuit of the current graph/digraph
hamiltonian_path()	Return a Hamiltonian path of the current graph/digraph
multicommodity_flow()	Solve a multicommodity flow problem.
disjoint_routed_paths()	Return a set of disjoint routed paths.
dominating_set()	Return a minimum dominating set of the graph
subgraph_search()	Return a copy of G in self.
subgraph_search_count()	Return the number of labelled occurrences of G in self.
subgraph_search_iterator()	Return an iterator over the labelled copies of G in self.
characteristic_polynomial()	Return the characteristic polynomial of the adjacency matrix of the (di)graph.
genus()	Return the minimal genus of the graph.
crossing_number()	Return the crossing number of the graph.

1.1.1 Methods

class sage.graphs.generic_graph.**GenericGraph**
Bases: *sage.graphs.generic_graph_pyx.GenericGraph_pyx*

Base class for graphs and digraphs.

__eq__(*other*)
Compare self and other for equality.

Do not call this method directly. That is, for `G.__eq__(H)` write `G == H`.

Two graphs are considered equal if the following hold:

- they are either both directed, or both undirected;
- they have the same settings for loops, multiedges, and weightedness;
- they have the same set of vertices;
- they have the same (multi)set of arrows/edges, where labels of arrows/edges are taken into account if *and only if* the graphs are considered weighted. See *weighted()*.

Note that this is *not* an isomorphism test.

EXAMPLES:

```
sage: G = graphs.EmptyGraph()
sage: H = Graph()
sage: G == H
True
sage: G.to_directed() == H.to_directed()
True
sage: G = graphs.RandomGNP(8, .9999)
sage: H = graphs.CompleteGraph(8)
sage: G == H # most often true
True
sage: G = Graph({0: [1, 2, 3, 4, 5, 6, 7]} )
sage: H = Graph({1: [0], 2: [0], 3: [0], 4: [0], 5: [0], 6: [0], 7: [0]} )
sage: G == H
True
sage: G.allow_loops(True)
sage: G == H
False
sage: G = graphs.RandomGNP(9, .3).to_directed()
sage: H = graphs.RandomGNP(9, .3).to_directed()
sage: G == H # most often false
False
sage: G = Graph(multiedges=True, sparse=True)
sage: G.add_edge(0, 1)
sage: H = copy(G)
sage: H.add_edge(0, 1)
sage: G == H
False
```

Note that graphs must be considered weighted, or Sage will not pay attention to edge label data in equality testing:

```
sage: foo = Graph(sparse=True)
sage: foo.add_edges([(0, 1, 1), (0, 2, 2)])
sage: bar = Graph(sparse=True)
```

(continues on next page)

(continued from previous page)

```
sage: bar.add_edges([(0, 1, 2), (0, 2, 1)])
sage: foo == bar
True
sage: foo.weighted(True)
sage: foo == bar
False
sage: bar.weighted(True)
sage: foo == bar
False
```

add_clique(*vertices*, *loops=False*)

Add a clique to the graph with the given vertices.

If the vertices are already present, only the edges are added.

INPUT:

- `vertices` – an iterable container of vertices for the clique to be added, e.g. a list, set, graph, etc.
- `loops` – boolean (default: `False`); whether to add edges from every given vertex to itself. This is allowed only if the (di)graph allows loops.

EXAMPLES:

```
sage: G = Graph()
sage: G.add_clique(range(4))
sage: G.is_isomorphic(graphs.CompleteGraph(4))
True
sage: D = DiGraph()
sage: D.add_clique(range(4))
sage: D.is_isomorphic(digraphs.Complete(4))
True
sage: D = DiGraph(loops=True)
sage: D.add_clique(range(4), loops=True)
sage: D.is_isomorphic(digraphs.Complete(4, loops=True))
True
sage: D = DiGraph(loops=False)
sage: D.add_clique(range(4), loops=True)
Traceback (most recent call last):
...
ValueError: cannot add edge from 0 to 0 in graph without loops
```

If the list of vertices contains repeated elements, a loop will be added at that vertex, even if `loops=False`:

```
sage: G = Graph(loops=True)
sage: G.add_clique([1, 1])
sage: G.edges()
[(1, 1, None)]
```

This is equivalent to:

```
sage: G = Graph(loops=True)
sage: G.add_clique([1], loops=True)
sage: G.edges()
[(1, 1, None)]
```

add_cycle(*vertices*)

Add a cycle to the graph with the given vertices.

If the vertices are already present, only the edges are added.

For digraphs, adds the directed cycle, whose orientation is determined by the list. Adds edges `(vertices[u], vertices[u+1])` and `(vertices[-1], vertices[0])`.

INPUT:

- `vertices` – an ordered list of the vertices of the cycle to be added

EXAMPLES:

```
sage: G = Graph()
sage: G.add_vertices(range(10)); G
Graph on 10 vertices
sage: show(G)
sage: G.add_cycle(list(range(10, 20)))
sage: show(G)
sage: G.add_cycle(list(range(10)))
sage: show(G)
```

```
sage: D = DiGraph()
sage: D.add_cycle(list(range(4)))
sage: D.edges()
[(0, 1, None), (1, 2, None), (2, 3, None), (3, 0, None)]
```

add_edge(*u*, *v=None*, *label=None*)

Add an edge from `u` to `v`.

INPUT: The following forms are all accepted:

- G.add_edge(1, 2)
- G.add_edge((1, 2))
- G.add_edges([(1, 2)])
- G.add_edge(1, 2, 'label')
- G.add_edge((1, 2, 'label'))
- G.add_edges([(1, 2, 'label')])

WARNING: The following intuitive input results in nonintuitive output:

```
sage: G = Graph()
sage: G.add_edge((1, 2), 'label')
sage: G.edges(sort=False)
[('label', (1, 2), None)]
```

You must either use the `label` keyword:

```
sage: G = Graph()
sage: G.add_edge((1, 2), label="label")
sage: G.edges(sort=False)
[(1, 2, 'label')]
```

Or use one of these:

```
sage: G = Graph()
sage: G.add_edge(1, 2, 'label')
sage: G.edges(sort=False)
[(1, 2, 'label')]
```

(continues on next page)

(continued from previous page)

```
sage: G = Graph()
sage: G.add_edge((1, 2, 'label'))
sage: G.edges(sort=False)
[(1, 2, 'label')]
```

Vertex name cannot be `None`, so:

```
sage: G = Graph()
sage: G.add_edge(None, 4)
sage: G.vertices()
[0, 4]
```

add_edges(*edges*, *loops=True*)

Add edges from an iterable container.

INPUT:

- `edges` – an iterable of edges, given either as `(u, v)` or `(u, v, label)`.
- `loops` – boolean (default: `True`); if `False`, remove all loops `(v, v)` from the input iterator. If `None`, remove loops unless the graph allows loops.

EXAMPLES:

```
sage: G = graphs.DodecahedralGraph()
sage: H = Graph()
sage: H.add_edges(G.edge_iterator()); H
Graph on 20 vertices
sage: G = graphs.DodecahedralGraph().to_directed()
sage: H = DiGraph()
sage: H.add_edges(G.edge_iterator()); H
Digraph on 20 vertices
sage: H.add_edges(iter([]))

sage: H = Graph()
sage: H.add_edges([(0, 1), (0, 2, "label")])
sage: H.edges()
[(0, 1, None), (0, 2, 'label')]
```

We demonstrate the `loops` argument:

```
sage: H = Graph()
sage: H.add_edges([(0, 0)], loops=False); H.edges()
[]
sage: H.add_edges([(0, 0)], loops=None); H.edges()
[]
sage: H.add_edges([(0, 0)]); H.edges()
Traceback (most recent call last):
...
ValueError: cannot add edge from 0 to 0 in graph without loops
sage: H = Graph(loops=True)
sage: H.add_edges([(0, 0)], loops=False); H.edges()
[]
sage: H.add_edges([(0, 0)], loops=None); H.edges()
[(0, 0, None)]
sage: H.add_edges([(0, 0)]); H.edges()
[(0, 0, None)]
```

add_path(*vertices*)
Add a path to the graph with the given vertices.

If the vertices are already present, only the edges are added.

For digraphs, adds the directed path `vertices[0], ..., vertices[-1]`.

INPUT:

- `vertices` – an ordered list of the vertices of the path to be added

EXAMPLES:

```
sage: G = Graph()
sage: G.add_vertices(range(10)); G
Graph on 10 vertices
sage: show(G)
sage: G.add_path(list(range(10, 20)))
sage: show(G)
sage: G.add_path(list(range(10)))
sage: show(G)
```

```
sage: D = DiGraph()
sage: D.add_path(list(range(4)))
sage: D.edges()
[(0, 1, None), (1, 2, None), (2, 3, None)]
```

add_vertex(*name=None*)
Create an isolated vertex.

If the vertex already exists, then nothing is done.

INPUT:

- `name` – an immutable object (default: `None`); when no name is specified (default), then the new vertex will be represented by the least integer not already representing a vertex. `name` must be an immutable object (e.g., an integer, a tuple, etc.).

As it is implemented now, if a graph G has a large number of vertices with numeric labels, then `G.add_vertex()` could potentially be slow, if `name=None`.

OUTPUT:

If `name=None`, the new vertex name is returned. `None` otherwise.

EXAMPLES:

```
sage: G = Graph(); G.add_vertex(); G
0
Graph on 1 vertex
```

```
sage: D = DiGraph(); D.add_vertex(); D
0
Digraph on 1 vertex
```

add_vertices(*vertices*)
Add vertices to the (di)graph from an iterable container of vertices.

Vertices that already exist in the graph will not be added again.

INPUT:

- `vertices` – iterator container of vertex labels. A new label is created, used and returned in the output list for all `None` values in `vertices`.

OUTPUT:

Generated names of new vertices if there is at least one `None` value present in `vertices`. `None` otherwise.

EXAMPLES:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7,8], 6: [8,
↪9], 7: [9]}
sage: G = Graph(d)
sage: G.add_vertices([10,11,12])
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: G.add_vertices(graphs.CycleGraph(25).vertex_iterator())
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
↪ 22, 23, 24]
```

```
sage: G = Graph()
sage: G.add_vertices([1, 2, 3])
sage: G.add_vertices([4, None, None, 5])
[0, 6]
```

adjacency_matrix(*sparse=None*, *vertices=None*)

Return the adjacency matrix of the (di)graph.

The matrix returned is over the integers. If a different ring is desired, use either the `sage.matrix.matrix0.Matrix.change_ring()` method or the `matrix()` function.

INPUT:

- `sparse` – boolean (default: `None`); whether to represent with a sparse matrix
- `vertices` – list (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *`GenericGraph.vertices()`* is used.

EXAMPLES:

```
sage: G = graphs.CubeGraph(4)
sage: G.adjacency_matrix()
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
```

```
sage: matrix(GF(2),G) # matrix over GF(2)
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
```

```
sage: D = DiGraph({0: [1, 2, 3], 1: [0, 2], 2: [3], 3: [4], 4: [0, 5], 5: [1]}
↪)
sage: D.adjacency_matrix()
[0 1 1 1 0 0]
[1 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 1]
[0 1 0 0 0 0]
```

A different ordering of the vertices:

```
sage: graphs.PathGraph(5).adjacency_matrix(vertices=[2, 4, 1, 3, 0])
[0 0 1 1 0]
[0 0 0 1 0]
[1 0 0 0 1]
[1 1 0 0 0]
[0 0 1 0 0]
```

all_paths(*G*, *start*, *end*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return the list of all paths between a pair of vertices.

If `start` is the same vertex as `end`, then `[[start]]` is returned – a list containing the 1-vertex, 0-edge path "`start`".

If `G` has multiple edges, a path will be returned as many times as the product of the multiplicity of the edges along that path depending on the value of the flag `use_multiedges`.

INPUT:

- `start` – a vertex of a graph, where to start
- `end` – a vertex of a graph, where to end
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`

  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)

- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

EXAMPLES:

```
sage: eg1 = Graph({0:[1, 2], 1:[4], 2:[3, 4], 4:[5], 5:[6]})
sage: eg1.all_paths(0, 6)
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg2 = graphs.PetersenGraph()
sage: sorted(eg2.all_paths(1, 4))
[[1, 0, 4],
 [1, 0, 5, 7, 2, 3, 4],
 [1, 0, 5, 7, 2, 3, 8, 6, 9, 4],
 [1, 0, 5, 7, 9, 4],
 [1, 0, 5, 7, 9, 6, 8, 3, 4],
 [1, 0, 5, 8, 3, 2, 7, 9, 4],
 [1, 0, 5, 8, 3, 4],
 [1, 0, 5, 8, 6, 9, 4],
 [1, 0, 5, 8, 6, 9, 7, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 8, 5, 0, 4],
 [1, 2, 3, 8, 5, 7, 9, 4],
 [1, 2, 3, 8, 6, 9, 4],
 [1, 2, 3, 8, 6, 9, 7, 5, 0, 4],
 [1, 2, 7, 5, 0, 4],
 [1, 2, 7, 5, 8, 3, 4],
 [1, 2, 7, 5, 8, 6, 9, 4],
 [1, 2, 7, 9, 4],
 [1, 2, 7, 9, 6, 8, 3, 4],
 [1, 2, 7, 9, 6, 8, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 9, 4],
 [1, 6, 8, 3, 4],
 [1, 6, 8, 5, 0, 4],
 [1, 6, 8, 5, 7, 2, 3, 4],
 [1, 6, 8, 5, 7, 9, 4],
 [1, 6, 9, 4],
 [1, 6, 9, 7, 2, 3, 4],
 [1, 6, 9, 7, 2, 3, 8, 5, 0, 4],
 [1, 6, 9, 7, 5, 0, 4],
 [1, 6, 9, 7, 5, 8, 3, 4]]
sage: dg = DiGraph({0:[1, 3], 1:[3], 2:[0, 3]})
sage: sorted(dg.all_paths(0, 3))
[[0, 1, 3], [0, 3]]
sage: ug = dg.to_undirected()
sage: sorted(ug.all_paths(0, 3))
[[0, 1, 3], [0, 2, 3], [0, 3]]

sage: g = Graph([(0, 1), (0, 1), (1, 2), (1, 2)], multiedges=True)
sage: g.all_paths(0, 2, use_multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

(continues on next page)

(continued from previous page)

```
sage: dg = DiGraph({0:[1, 2, 1], 3:[0, 0]}, multiedges=True)
sage: dg.all_paths(3, 1, use_multiedges=True)
[[3, 0, 1], [3, 0, 1], [3, 0, 1], [3, 0, 1]]

sage: g = Graph([(0, 1, 'a'), (0, 1, 'b'), (1, 2, 'c'), (1, 2, 'd')],␣
↪multiedges=True)
sage: g.all_paths(0, 2, use_multiedges=False)
[[0, 1, 2]]
sage: g.all_paths(0, 2, use_multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=True)
[[(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)]]
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=True, labels=True)
[((0, 1, 'b'), (1, 2, 'd')),
 ((0, 1, 'b'), (1, 2, 'c')),
 ((0, 1, 'a'), (1, 2, 'd')),
 ((0, 1, 'a'), (1, 2, 'c'))]
sage: g.all_paths(0, 2, use_multiedges=False, report_edges=True, labels=True)
[((0, 1, 'b'), (1, 2, 'd'))]
sage: g.all_paths(0, 2, use_multiedges=False, report_edges=False, labels=True)
[[0, 1, 2]]
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=False, labels=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

allow_loops (*new*, *check=True*)

Change whether loops are permitted in the (di)graph

INPUT:

- `new` – boolean
- `check` – boolean (default: `True`); whether to remove existing loops from the (di)graph when the new status is `False`

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0, 0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
```

(continues on next page)

(continued from previous page)

```
sage: D.allows_loops()
True
sage: D.add_edge((0, 0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

allow_multiple_edges (*new*, *check=True*, *keep_label='any'*)

Change whether multiple edges are permitted in the (di)graph.

INPUT:

- `new` – boolean; if `True`, the new graph will allow multiple edges
- `check` – boolean (default: `True`); if `True` and `new` is `False`, we remove all multiple edges from the graph
- `keep_label` – string (default: `'any'`); used only if `new` is `False` and `check` is `True`. If there are multiple edges with different labels, this variable defines which label should be kept:
  - `'any'` – any label
  - `'min'` – the smallest label
  - `'max'` – the largest label

Warning: `'min'` and `'max'` only works if the labels can be compared. A `TypeError` might be raised when working with non-comparable objects in Python 3.

EXAMPLES:

The standard behavior with undirected graphs:

```
sage: G = Graph(multiedges=True, sparse=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0, 1, 1), (0, 1, 2), (0, 1, 3)])
sage: G.has_multiple_edges()
True
sage: G.multiple_edges(sort=True)
[(0, 1, 1), (0, 1, 2), (0, 1, 3)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, 3)]
```

If we ask for the minimum label:

```
sage: G = Graph([(0, 1, 1), (0, 1, 2), (0, 1, 3)], multiedges=True,␣
↪sparse=True)
sage: G.allow_multiple_edges(False, keep_label='min')
sage: G.edges()
[(0, 1, 1)]
```

If we ask for the maximum label:

```
sage: G = Graph([(0, 1, 1), (0, 1, 2), (0, 1, 3)], multiedges=True,␣
↪sparse=True)
sage: G.allow_multiple_edges(False, keep_label='max')
sage: G.edges()
[(0, 1, 3)]
```

The standard behavior with digraphs:

```
sage: D = DiGraph(multiedges=True, sparse=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0, 1)] * 3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]
```

allows_loops()

Return whether loops are permitted in the (di)graph

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0, 0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]
```

(continues on next page)

(continued from previous page)

```
sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0, 0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

allows_multiple_edges()

Return whether multiple edges are permitted in the (di)graph.

EXAMPLES:

```
sage: G = Graph(multiedges=True, sparse=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0, 1)] * 3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True, sparse=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0, 1)] * 3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
```

(continues on next page)

(continued from previous page)

```
sage: D.edges()
[(0, 1, None)]
```

am(*sparse=None*, *vertices=None*)

Return the adjacency matrix of the (di)graph.

The matrix returned is over the integers. If a different ring is desired, use either the `sage.matrix.matrix0.Matrix.change_ring()` method or the `matrix()` function.

INPUT:

- `sparse` – boolean (default: `None`); whether to represent with a sparse matrix
- `vertices` – list (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *`GenericGraph.vertices()`* is used.

EXAMPLES:

```
sage: G = graphs.CubeGraph(4)
sage: G.adjacency_matrix()
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
```

```
sage: matrix(GF(2),G) # matrix over GF(2)
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]
```

```
sage: D = DiGraph({0: [1, 2, 3], 1: [0, 2], 2: [3], 3: [4], 4: [0, 5], 5: [1]}
↪)
```

(continues on next page)

(continued from previous page)

```
sage: D.adjacency_matrix()
[0 1 1 1 0 0]
[1 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 1]
[0 1 0 0 0 0]
```

A different ordering of the vertices:

```
sage: graphs.PathGraph(5).adjacency_matrix(vertices=[2, 4, 1, 3, 0])
[0 0 1 1 0]
[0 0 0 1 0]
[1 0 0 0 1]
[1 1 0 0 0]
[0 0 1 0 0]
```

antisymmetric()

Check whether the graph is antisymmetric.

A graph represents an antisymmetric relation if the existence of a path from a vertex x to a vertex y implies that there is not a path from y to x unless $x = y$.

EXAMPLES:

A directed acyclic graph is antisymmetric:

```
sage: G = digraphs.RandomDirectedGNR(20, 0.5)
sage: G.antisymmetric()
True
```

Loops are allowed:

```
sage: G.allow_loops(True)
sage: G.add_edge(0, 0)
sage: G.antisymmetric()
True
```

An undirected graph is never antisymmetric unless it is just a union of isolated vertices (with possible loops):

```
sage: graphs.RandomGNP(20, 0.5).antisymmetric()
False
sage: Graph(3).antisymmetric()
True
sage: Graph([(i, i) for i in range(3)], loops=True).antisymmetric()
True
sage: DiGraph([(i, i) for i in range(3)], loops=True).antisymmetric()
True
```

automorphism_group(*partition=None*, *verbosity=0*, *edge_labels=False*, *order=False*, *return_group=True*, *orbits=False*, *algorithm=None*)

Return the automorphism group of the graph.

With `partition` this can also return the largest subgroup of the automorphism group of the (di)graph whose orbit partition is finer than the partition given.

INPUT:

- `partition` - default is the unit partition, otherwise computes the subgroup of the full automorphism group respecting the partition.
- `edge_labels` - default False, otherwise allows only permutations respecting edge labels.
- `order` - (default False) if True, compute the order of the automorphism group
- `return_group` - default True
- `orbits` - returns the orbits of the group acting on the vertices of the graph
- `algorithm` - If `algorithm = "bliss"` the automorphism group is computed using the optional package bliss (http://www.tcs.tkk.fi/Software/bliss/index.html). Setting it to "sage" uses Sage's implementation. If set to `None` (default), bliss is used when available.

OUTPUT: The order of the output is group, order, orbits. However, there are options to turn each of these on or off.

EXAMPLES:

Graphs:

```
sage: graphs_query = GraphQuery(display_cols=['graph6'],num_vertices=4)
sage: L = graphs_query.get_graphs_list()
sage: graphs_list.show_graphs(L)
sage: for g in L:
....:     G = g.automorphism_group()
....:     G.order(), G.gens()
(24, [(2,3), (1,2), (0,1)])
(4, [(2,3), (0,1)])
(2, [(1,2)])
(6, [(1,2), (0,1)])
(6, [(2,3), (1,2)])
(8, [(1,2), (0,1)(2,3)])
(2, [(0,1)(2,3)])
(2, [(1,2)])
(8, [(2,3), (0,1), (0,2)(1,3)])
(4, [(2,3), (0,1)])
(24, [(2,3), (1,2), (0,1)])
sage: C = graphs.CubeGraph(4)
sage: G = C.automorphism_group()
sage: M = G.character_table() # random order of rows, thus abs() below
sage: QQ(M.determinant()).abs()
712483534798848
sage: G.order()
384
```

```
sage: D = graphs.DodecahedralGraph()
sage: G = D.automorphism_group()
sage: A5 = AlternatingGroup(5)
sage: Z2 = CyclicPermutationGroup(2)
sage: H = A5.direct_product(Z2)[0] #see documentation for direct_product to␣
↪explain the [0]
sage: G.is_isomorphic(H)
True
```

Multigraphs:

```
sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edge(('a', 'b'))
```

(continues on next page)

(continued from previous page)

```
sage: G.add_edge(('a', 'b'))
sage: G.add_edge(('a', 'b'))
sage: G.automorphism_group()
Permutation Group with generators [('a','b')]
```

Digraphs:

```
sage: D = DiGraph( { 0:[1], 1:[2], 2:[3], 3:[4], 4:[0] } )
sage: D.automorphism_group()
Permutation Group with generators [(0,1,2,3,4)]
```

Edge labeled graphs:

```
sage: G = Graph(sparse=True)
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: G.automorphism_group(edge_labels=True)
Permutation Group with generators [(1,4)(2,3)]

sage: G.automorphism_group(edge_labels=True, algorithm="bliss") # optional -␣
↪bliss
Permutation Group with generators [(1,4)(2,3)]

sage: G.automorphism_group(edge_labels=True, algorithm="sage")
Permutation Group with generators [(1,4)(2,3)]
```

```
sage: G = Graph({0 : {1 : 7}})
sage: G.automorphism_group(edge_labels=True)
Permutation Group with generators [(0,1)]

sage: foo = Graph(sparse=True)
sage: bar = Graph(sparse=True)
sage: foo.add_edges([(0,1,1),(1,2,2), (2,3,3)])
sage: bar.add_edges([(0,1,1),(1,2,2), (2,3,3)])
sage: foo.automorphism_group(edge_labels=True)
Permutation Group with generators [()]
sage: foo.automorphism_group()
Permutation Group with generators [(0,3)(1,2)]
sage: bar.automorphism_group(edge_labels=True)
Permutation Group with generators [()]
```

You can also ask for just the order of the group:

```
sage: G = graphs.PetersenGraph()
sage: G.automorphism_group(return_group=False, order=True)
120
```

Or, just the orbits (note that each graph here is vertex transitive)

```
sage: G = graphs.PetersenGraph()
sage: G.automorphism_group(return_group=False, orbits=True,algorithm='sage')
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
sage: orb = G.automorphism_group(partition=[[0],list(range(1,10))],
....:                            return_group=False, orbits=True,algorithm=
↪'sage')
sage: sorted([sorted(o) for o in orb], key=len)
[[0], [1, 4, 5], [2, 3, 6, 7, 8, 9]]
```

(continues on next page)

(continued from previous page)

```
sage: C = graphs.CubeGraph(3)
sage: orb = C.automorphism_group(orbits=True, return_group=False,algorithm=
↪'sage')
sage: [sorted(o) for o in orb]
[['000', '001', '010', '011', '100', '101', '110', '111']]
```

One can also use the faster algorithm for computing the automorphism group of the graph - bliss:

```
sage: G = graphs.HallJankoGraph()                    # optional - bliss
sage: A1 = G.automorphism_group()                    # optional - bliss
sage: A2 = G.automorphism_group(algorithm='bliss')   # optional - bliss
sage: A1.is_isomorphic(A2)                           # optional - bliss
True
```

average_degree()

Return the average degree of the graph.

The average degree of a graph $G = (V, E)$ is equal to $\frac{2|E|}{|V|}$.

EXAMPLES:

The average degree of a regular graph is equal to the degree of any vertex:

```
sage: g = graphs.CompleteGraph(5)
sage: g.average_degree() == 4
True
```

The average degree of a tree is always strictly less than 2:

```
sage: tree = graphs.RandomTree(20)
sage: tree.average_degree() < 2
True
```

For any graph, it is equal to $\frac{2|E|}{|V|}$:

```
sage: g = graphs.RandomGNP(20, .4)
sage: g.average_degree() == 2 * g.size() / g.order()
True
```

average_distance(*by_weight=False*, *algorithm=None*, *weight_function=None*)

Return the average distance between vertices of the graph.

Formally, for a graph G this value is equal to $\frac{1}{n(n-1)} \sum_{u,v \in G} d(u, v)$ where $d(u, v)$ denotes the distance between vertices u and v and n is the number of vertices in G.

For more information on the input variables and more examples, we refer to *wiener_index()* and *shortest_path_all_pairs()*, which have very similar input variables.

INPUT:

- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); one of the algorithms available for method *wiener_index()*
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.

- `check_weight` – boolean (default: `True`); if `True`, we check that the weight_function outputs a number for each edge

EXAMPLES:

From [GYLL1993]:

```
sage: g=graphs.PathGraph(10)
sage: w=lambda x: (x*(x*x -1)/6)/(x*(x-1)/2)
sage: g.average_distance()==w(10)
True
```

blocks_and_cut_vertices(*G*, *algorithm='Tarjan_Boost'*, *sort=False*)

Return the blocks and cut vertices of the graph.

In the case of a digraph, this computation is done on the underlying graph.

A cut vertex is one whose deletion increases the number of connected components. A block is a maximal induced subgraph which itself has no cut vertices. Two distinct blocks cannot overlap in more than a single cut vertex.

INPUT:

- `algorithm` – string (default: `"Tarjan_Boost"`); the algorithm to use among:
  - `"Tarjan_Boost"` (default) – Tarjan's algorithm (Boost implementation)
  - `"Tarjan_Sage"` – Tarjan's algorithm (Sage implementation)
- `sort` – boolean (default: `False`); whether to sort vertices inside the components and the list of cut vertices **currently only available for ‘“Tarjan_Sage”‘**

OUTPUT: `(B, C)`, where `B` is a list of blocks - each is a list of vertices and the blocks are the corresponding induced subgraphs - and `C` is a list of cut vertices.

ALGORITHM:

We implement the algorithm proposed by Tarjan in [Tarjan72]. The original version is recursive. We emulate the recursion using a stack.

See also:

- *blocks_and_cuts_tree()*
- *sage.graphs.base.boost_graph.blocks_and_cut_vertices()*
- *is_biconnected()*
- *bridges()*

EXAMPLES:

We construct a trivial example of a graph with one cut vertex:

```
sage: from sage.graphs.connectivity import blocks_and_cut_vertices
sage: rings = graphs.CycleGraph(10)
sage: rings.merge_vertices([0, 5])
sage: blocks_and_cut_vertices(rings)
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
sage: rings.blocks_and_cut_vertices()
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
sage: B, C = blocks_and_cut_vertices(rings, algorithm="Tarjan_Sage",␣
↪sort=True)
```

(continues on next page)

(continued from previous page)

```
sage: B, C
([[0, 1, 2, 3, 4], [0, 6, 7, 8, 9]], [0])
sage: B2, C2 = blocks_and_cut_vertices(rings, algorithm="Tarjan_Sage",␣
↪sort=False)
sage: Set(map(Set, B)) == Set(map(Set, B2)) and set(C) == set(C2)
True
```

The Petersen graph is biconnected, hence has no cut vertices:

```
sage: blocks_and_cut_vertices(graphs.PetersenGraph())
([[0, 1, 4, 5, 2, 6, 3, 7, 8, 9]], [])
```

Decomposing paths to pairs:

```
sage: g = graphs.PathGraph(4) + graphs.PathGraph(5)
sage: blocks_and_cut_vertices(g)
([[2, 3], [1, 2], [0, 1], [7, 8], [6, 7], [5, 6], [4, 5]], [1, 2, 5, 6, 7])
```

A disconnected graph:

```
sage: g = Graph({1: {2: 28, 3: 10}, 2: {1: 10, 3: 16}, 4: {}, 5: {6: 3, 7: 10,
↪ 8: 4}})
sage: blocks_and_cut_vertices(g)
([[1, 2, 3], [5, 6], [5, 7], [5, 8], [4]], [5])
```

A directed graph with Boost's algorithm (trac ticket #25994):

```
sage: rings = graphs.CycleGraph(10)
sage: rings.merge_vertices([0, 5])
sage: rings = rings.to_directed()
sage: blocks_and_cut_vertices(rings, algorithm="Tarjan_Boost")
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
```

blocks_and_cuts_tree(*G*)

Return the blocks-and-cuts tree of `self`.

This new graph has two different kinds of vertices, some representing the blocks (type B) and some other the cut vertices of the graph (type C).

There is an edge between a vertex u of type B and a vertex v of type C if the cut-vertex corresponding to v is in the block corresponding to u.

The resulting graph is a tree, with the additional characteristic property that the distance between two leaves is even. When `self` is not connected, the resulting graph is a forest.

When `self` is biconnected, the tree is reduced to a single node of type B.

We referred to [HarPri] and [Gallai] for blocks and cuts tree.

See also:

- *blocks_and_cut_vertices()*
- *is_biconnected()*

EXAMPLES:

```
sage: from sage.graphs.connectivity import blocks_and_cuts_tree
sage: T = blocks_and_cuts_tree(graphs.KrackhardtKiteGraph()); T
Graph on 5 vertices
sage: T.is_isomorphic(graphs.PathGraph(5))
True
sage: from sage.graphs.connectivity import blocks_and_cuts_tree
sage: T = graphs.KrackhardtKiteGraph().blocks_and_cuts_tree(); T
Graph on 5 vertices
```

The distance between two leaves is even:

```
sage: T = blocks_and_cuts_tree(graphs.RandomTree(40))
sage: T.is_tree()
True
sage: leaves = [v for v in T if T.degree(v) == 1]
sage: all(T.distance(u,v) % 2 == 0 for u in leaves for v in leaves)
True
```

The tree of a biconnected graph has a single vertex, of type B:

```
sage: T = blocks_and_cuts_tree(graphs.PetersenGraph())
sage: T.vertices()
[('B', (0, 1, 4, 5, 2, 6, 3, 7, 8, 9))]
```

breadth_first_search(*start*, *ignore_direction=False*, *distance=None*, *neighbors=None*, *report_distance=False*, *edges=False*)

Return an iterator over the vertices in a breadth-first ordering.

INPUT:

- `start` – vertex or list of vertices from which to start the traversal
- `ignore_direction` – boolean (default `False`); only applies to directed graphs. If `True`, searches across edges in either direction.
- `distance` – integer (default: `None`); the maximum distance from the `start` nodes to traverse. The `start` nodes are at distance zero from themselves.
- `neighbors` – function (default: `None`); a function that inputs a vertex and return a list of vertices. For an undirected graph, `neighbors` is by default the *neighbors()* function. For a digraph, the `neighbors` function defaults to the *neighbor_out_iterator()* function of the graph.
- `report_distance` – boolean (default `False`); if `True`, reports pairs `(vertex, distance)` where `distance` is the distance from the `start` nodes. If `False` only the vertices are reported.
- `edges` – boolean (default `False`); whether to return the edges of the BFS tree in the order of visit or the vertices (default). Edges are directed in root to leaf orientation of the tree.

  Note that parameters `edges` and `report_distance` cannot be `True` simultaneously.

See also:

- *breadth_first_search* – breadth-first search for fast compiled graphs.
- *depth_first_search* – depth-first search for fast compiled graphs.
- *depth_first_search()* – depth-first search for generic graphs.

EXAMPLES:

```
sage: G = Graph({0: [1], 1: [2], 2: [3], 3: [4], 4: [0]})
sage: list(G.breadth_first_search(0))
[0, 1, 4, 2, 3]
```

By default, the edge direction of a digraph is respected, but this can be overridden by the `ignore_direction` parameter:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.breadth_first_search(0))
[0, 1, 2, 3, 4, 5, 6, 7]
sage: list(D.breadth_first_search(0, ignore_direction=True))
[0, 1, 2, 3, 7, 4, 5, 6]
```

You can specify a maximum distance in which to search. A distance of zero returns the `start` vertices:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.breadth_first_search(0, distance=0))
[0]
sage: list(D.breadth_first_search(0, distance=1))
[0, 1, 2, 3]
```

Multiple starting vertices can be specified in a list:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.breadth_first_search([0]))
[0, 1, 2, 3, 4, 5, 6, 7]
sage: list(D.breadth_first_search([0, 6]))
[0, 6, 1, 2, 3, 7, 4, 5]
sage: list(D.breadth_first_search([0, 6], distance=0))
[0, 6]
sage: list(D.breadth_first_search([0, 6], distance=1))
[0, 6, 1, 2, 3, 7]
sage: list(D.breadth_first_search(6, ignore_direction=True, distance=2))
[6, 3, 7, 0, 5]
```

More generally, you can specify a `neighbors` function. For example, you can traverse the graph backwards by setting `neighbors` to be the *neighbors_in()* function of the graph:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.breadth_first_search(5, neighbors=D.neighbors_in, distance=2))
[5, 1, 2, 0]
sage: list(D.breadth_first_search(5, neighbors=D.neighbors_out, distance=2))
[5, 7, 0]
sage: list(D.breadth_first_search(5 ,neighbors=D.neighbors, distance=2))
[5, 1, 2, 7, 0, 4, 6]
```

It is possible (trac ticket #16470) using the keyword `report_distance` to get pairs `(vertex, distance)` encoding the distance from the starting vertices:

```
sage: G = graphs.PetersenGraph()
sage: list(G.breadth_first_search(0, report_distance=True))
[(0, 0), (1, 1), (4, 1), (5, 1), (2, 2), (6, 2), (3, 2), (9, 2),
(7, 2), (8, 2)]
```

(continues on next page)

(continued from previous page)

```
sage: list(G.breadth_first_search(0, report_distance=False))
[0, 1, 4, 5, 2, 6, 3, 9, 7, 8]

sage: D = DiGraph({0: [1, 3], 1: [0, 2], 2: [0, 3], 3: [4]})
sage: D.show()
sage: list(D.breadth_first_search(4, neighbors=D.neighbor_in_iterator, report_
↪distance=True))
[(4, 0), (3, 1), (0, 2), (2, 2), (1, 3)]

sage: C = graphs.CycleGraph(4)
sage: list(C.breadth_first_search([0, 1], report_distance=True))
[(0, 0), (1, 0), (3, 1), (2, 1)]
```

You can get edges of the BFS tree instead of the vertices using the `edges` parameter:

```
sage: D = DiGraph({1:[2,3],2:[4],3:[4],4:[1],5:[2,6]})
sage: list(D.breadth_first_search(1, edges=True))
[(1, 2), (1, 3), (2, 4)]
```

canonical_label(*partition=None*, *certificate=False*, *edge_labels=False*, *algorithm=None*, *return_graph=True*)

Return the canonical graph.

A canonical graph is the representative graph of an isomorphism class by some canonization function c. If G and H are graphs, then $G \cong c(G)$, and $c(G) == c(H)$ if and only if $G \cong H$.

See the Wikipedia article Graph_canonization for more information.

INPUT:

- `partition` – if given, the canonical label with respect to this set partition will be computed. The default is the unit set partition.
- `certificate` – boolean (default: `False`). When set to `True`, a dictionary mapping from the vertices of the (di)graph to its canonical label will also be returned.
- `edge_labels` – boolean (default: `False`). When set to `True`, allows only permutations respecting edge labels.
- `algorithm` – a string (default: `None`). The algorithm to use; currently available:
  - `'bliss'`: use the optional package bliss (http://www.tcs.tkk.fi/Software/bliss/index.html);
  - `'sage'`: always use Sage's implementation.
  - `None` (default): use bliss when available and possible

  Note: Make sure you always compare canonical forms obtained by the same algorithm.

- `return_graph` – boolean (default: `True`). When set to `False`, returns the list of edges of the canonical graph instead of the canonical graph; only available when `'bliss'` is explicitly set as algorithm.

EXAMPLES:

Canonization changes isomorphism to equality:

```
sage: g1 = graphs.GridGraph([2,3])
sage: g2 = Graph({1: [2, 4], 3: [2, 6], 5: [4, 2, 6]})
sage: g1 == g2
False
sage: g1.is_isomorphic(g2)
True
sage: g1.canonical_label() == g2.canonical_label()
True
```

We can get the relabeling used for canonization:

```
sage: g, c = g1.canonical_label(algorithm='sage', certificate=True)
sage: g
Grid Graph for [2, 3]: Graph on 6 vertices
sage: c  # py2
{(0, 0): 2, (0, 1): 4, (0, 2): 3, (1, 0): 1, (1, 1): 5, (1, 2): 0}
sage: c  # py3
{(0, 0): 3, (0, 1): 4, (0, 2): 2, (1, 0): 0, (1, 1): 5, (1, 2): 1}
```

Multigraphs and directed graphs work too:

```
sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edge((0,1))
sage: G.add_edge((0,1))
sage: G.add_edge((0,1))
sage: G.canonical_label()
Multi-graph on 2 vertices
sage: Graph('A?').canonical_label()
Graph on 2 vertices

sage: P = graphs.PetersenGraph()
sage: DP = P.to_directed()
sage: DP.canonical_label(algorithm='sage').adjacency_matrix()
[0 0 0 0 0 0 0 1 1 1]
[0 0 0 0 1 0 1 0 0 1]
[0 0 0 1 0 0 1 0 1 0]
[0 0 1 0 0 1 0 0 0 1]
[0 1 0 0 0 1 0 0 1 0]
[0 0 0 1 1 0 0 1 0 0]
[0 1 1 0 0 0 0 1 0 0]
[1 0 0 0 0 1 1 0 0 0]
[1 0 1 0 1 0 0 0 0 0]
[1 1 0 1 0 0 0 0 0 0]
```

Edge labeled graphs:

```
sage: G = Graph(sparse=True)
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: G.canonical_label(edge_labels=True)
Graph on 5 vertices
sage: G.canonical_label(edge_labels=True, algorithm="bliss",␣
↪certificate=True) # optional - bliss
(Graph on 5 vertices, {0: 4, 1: 3, 2: 1, 3: 0, 4: 2})

sage: G.canonical_label(edge_labels=True, algorithm="sage", certificate=True)
(Graph on 5 vertices, {0: 4, 1: 3, 2: 0, 3: 1, 4: 2})
```

Another example where different canonization algorithms give different graphs:

```
sage: g = Graph({'a': ['b'], 'c': ['d']})
sage: g_sage = g.canonical_label(algorithm='sage')
sage: g_bliss = g.canonical_label(algorithm='bliss')  # optional - bliss
sage: g_sage.edges(labels=False)
[(0, 3), (1, 2)]
sage: g_bliss.edges(labels=False)  # optional - bliss
[(0, 1), (2, 3)]
```

cartesian_product(*other*)

Return the Cartesian product of `self` and `other`.

The Cartesian product of G and H is the graph L with vertex set $V(L)$ equal to the Cartesian product of the vertices $V(G)$ and $V(H)$, and $((u, v), (w, x))$ is an edge iff either - (u, w) is an edge of self and $v = x$, or - (v, x) is an edge of other and $u = w$.

See also:

- *is_cartesian_product()* – factorization of graphs according to the Cartesian product
- *graph_products* – a module on graph products

categorical_product(*other*)

Return the tensor product of `self` and `other`.

The tensor product of G and H is the graph L with vertex set $V(L)$ equal to the Cartesian product of the vertices $V(G)$ and $V(H)$, and $((u, v), (w, x))$ is an edge iff - (u, w) is an edge of self, and - (v, x) is an edge of other.

The tensor product is also known as the categorical product and the kronecker product (refering to the kronecker matrix product). See the Wikipedia article Kronecker_product.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: T = C.tensor_product(Z); T
Graph on 10 vertices
sage: T.size()
10
sage: T.plot() # long time
Graphics object consisting of 21 graphics primitives
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: T = D.tensor_product(P); T
Graph on 200 vertices
sage: T.size()
900
sage: T.plot() # long time
Graphics object consisting of 1101 graphics primitives
```

center(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the set of vertices in the center of the (di)graph.

The center is the set of vertices whose eccentricity is equal to the radius of the (di)graph, i.e., achieving the minimum eccentricity.

For more information and examples on how to use input variables, see *shortest_paths()* and *eccentricity()*

INPUT:

- by_weight – boolean (default: False); if True, edge weights are taken into account; if False, all edges have weight 1
- algorithm – string (default: None); see method *eccentricity()* for the list of available algorithms
- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

Is Central African Republic in the center of Africa in graph theoretic sense? Yes:

```
sage: A = graphs.AfricaMap(continental=True)
sage: sorted(A.center())
['Cameroon', 'Central Africa']
```

Some other graphs. Center can be the whole graph:

```
sage: G = graphs.DiamondGraph()
sage: G.center()
[1, 2]
sage: P = graphs.PetersenGraph()
sage: P.subgraph(P.center()) == P
True
sage: S = graphs.StarGraph(19)
sage: S.center()
[0]
```

centrality_betweenness(*k=None*, *normalized=True*, *weight=None*, *endpoints=False*, *seed=None*, *exact=False*, *algorithm=None*)

Return the betweenness centrality.

The betweenness centrality of a vertex is the fraction of number of shortest paths that go through each vertex. The betweenness is normalized by default to be in range (0,1).

Measures of the centrality of a vertex within a graph determine the relative importance of that vertex to its graph. Vertices that occur on more shortest paths between other vertices have higher betweenness than vertices that occur on less.

INPUT:

- normalized – boolean (default: True); if set to False, result is not normalized.
- k – integer (default: None); if set to an integer, use k node samples to estimate betweenness. Higher values give better approximations. Not available when algorithm="Sage".
- weight – string (default: None); if set to a string, use that attribute of the nodes as weight. weight = True is equivalent to weight = "weight". Not available when algorithm="Sage".
- endpoints – boolean (default: False); if set to True it includes the endpoints in the shortest paths count. Not available when algorithm="Sage".
- exact – boolean (default: False); whether to compute over rationals or on double C variables. Not available when algorithm="NetworkX".

- `algorithm` – string (default: `None`); can be either `"Sage"` (see *centrality*), `"NetworkX"` or `"None"`. In the latter case, Sage's algorithm will be used whenever possible.

See also:

- *centrality_degree()*
- *centrality_closeness()*

EXAMPLES:

```
sage: g = graphs.ChvatalGraph()
sage: g.centrality_betweenness() # abs tol 1e-10
{0: 0.06969696969696969, 1: 0.06969696969696969,
 2: 0.0606060606060606, 3: 0.0606060606060606,
 4: 0.06969696969696969, 5: 0.06969696969696969,
 6: 0.0606060606060606, 7: 0.0606060606060606,
 8: 0.0606060606060606, 9: 0.0606060606060606,
 10: 0.0606060606060606, 11: 0.0606060606060606}
sage: g.centrality_betweenness(normalized=False) # abs tol 1e-10
{0: 3.833333333333333, 1: 3.833333333333333, 2: 3.333333333333333,
 3: 3.333333333333333, 4: 3.833333333333333, 5: 3.833333333333333,
 6: 3.333333333333333, 7: 3.333333333333333, 8: 3.333333333333333,
 9: 3.333333333333333, 10: 3.333333333333333,
 11: 3.333333333333333}
sage: D = DiGraph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: D.show(figsize=[2,2])
sage: D = D.to_undirected()
sage: D.show(figsize=[2,2])
sage: D.centrality_betweenness() # abs tol abs 1e-10
{0: 0.16666666666666666, 1: 0.16666666666666666, 2: 0.0, 3: 0.0}
```

centrality_closeness(*vert=None*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the closeness centrality of all vertices in `vert`.

In a (strongly) connected graph, the closeness centrality of a vertex v is equal to the inverse of the average distance between v and other vertices. If the graph is disconnected, the closeness centrality of v is multiplied by the fraction of reachable vertices in the graph: this way, central vertices should also reach several other vertices in the graph [OLJ2014]. In formulas,

$$c(v) = \frac{r(v) - 1}{\sum_{w \in R(v)} d(v, w)} \frac{r(v) - 1}{n - 1}$$

where $R(v)$ is the set of vertices reachable from v, and $r(v)$ is the cardinality of $R(v)$.

'Closeness centrality may be defined as the total graph-theoretic distance of a given vertex from all other vertices... Closeness is an inverse measure of centrality in that a larger value indicates a less central actor while a smaller value indicates a more central actor,' [Bor1995].

For more information, see the Wikipedia article Centrality.

INPUT:

- `vert` – the vertex or the list of vertices we want to analyze. If `None` (default), all vertices are considered.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, and otherwise all edges have weight 1
- `algorithm` – string (default: `None`); one of the following algorithms:

  - `'BFS'`: performs a BFS from each vertex that has to be analyzed. Does not work with edge weights.
  - `'NetworkX'`: the NetworkX algorithm (works only with positive weights).
  - `'Dijkstra_Boost'`: the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - `'Floyd-Warshall-Cython'`: the Cython implementation of the Floyd-Warshall algorithm. Works only if `by_weight==False` and all centralities are needed.
  - `'Floyd-Warshall-Python'`: the Python implementation of the Floyd-Warshall algorithm. Works only if all centralities are needed, but it can deal with weighted graphs, even with negative weights (but no negative cycle is allowed).
  - `'Johnson_Boost'`: the Johnson algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - `None` (default): Sage chooses the best algorithm: `'BFS'` if `by_weight` is `False`, `'Dijkstra_Boost'` if all weights are positive, `'Johnson_Boost'` otherwise.

- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); if `True`, we check that the `weight_function` outputs a number for each edge.

OUTPUT:

If `vert` is a vertex, the closeness centrality of that vertex. Otherwise, a dictionary associating to each vertex in `vert` its closeness centrality. If a vertex has (out)degree 0, its closeness centrality is not defined, and the vertex is not included in the output.

See also:

- *centrality_closeness_top_k()*
- *centrality_degree()*
- *centrality_betweenness()*

EXAMPLES:

Standard examples:

```
sage: (graphs.ChvatalGraph()).centrality_closeness()
{0: 0.61111111111111..., 1: 0.61111111111111..., 2: 0.61111111111111..., 3: 0.
↪61111111111111..., 4: 0.61111111111111..., 5: 0.61111111111111..., 6: 0.
↪61111111111111..., 7: 0.61111111111111..., 8: 0.61111111111111..., 9: 0.
↪61111111111111..., 10: 0.61111111111111..., 11: 0.61111111111111...}
sage: D = DiGraph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: D.show(figsize=[2,2])
sage: D.centrality_closeness(vert=[0,1])
{0: 1.0, 1: 0.3333333333333333}
sage: D = D.to_undirected()
sage: D.show(figsize=[2,2])
sage: D.centrality_closeness()
{0: 1.0, 1: 1.0, 2: 0.75, 3: 0.75}
```

In a (strongly) connected (di)graph, the closeness centrality of v is inverse of the average distance between v and all other vertices:

```
sage: g = graphs.PathGraph(5)
sage: g.centrality_closeness(0)
0.4
sage: dist = g.shortest_path_lengths(0).values()
sage: float(len(dist)-1) / sum(dist)
0.4
sage: d = g.to_directed()
sage: d.centrality_closeness(0)
0.4
sage: dist = d.shortest_path_lengths(0).values()
sage: float(len(dist)-1) / sum(dist)
0.4
```

If a vertex has (out)degree 0, its closeness centrality is not defined:

```
sage: g = Graph(5)
sage: g.centrality_closeness()
{}
sage: print(g.centrality_closeness(0))
None
```

Weighted graphs:

```
sage: D = graphs.GridGraph([2,2])
sage: weight_function = lambda e:10
sage: D.centrality_closeness([(0,0),(0,1)])                              # tol␣
↪abs 1e-12
{(0, 0): 0.75, (0, 1): 0.75}
sage: D.centrality_closeness((0,0), weight_function=weight_function) # tol␣
↪abs 1e-12
0.075
```

characteristic_polynomial(*var='x'*, *laplacian=False*)

Return the characteristic polynomial of the adjacency matrix of the (di)graph.

Let G be a (simple) graph with adjacency matrix A. Let I be the identity matrix of dimensions the same as A. The characteristic polynomial of G is defined as the determinant $\det(xI - A)$.

Note: `characteristic_polynomial` and `charpoly` are aliases and thus provide exactly the same method.

INPUT:

- `x` – (default: `'x'`); the variable of the characteristic polynomial
- `laplacian` – boolean (default: `False`); if `True`, use the Laplacian matrix

See also:

- *kirchhoff_matrix()*
- *laplacian_matrix()*

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.characteristic_polynomial()
```

(continues on next page)

(continued from previous page)

```
x^10 - 15*x^8 + 75*x^6 - 24*x^5 - 165*x^4 + 120*x^3 + 120*x^2 - 160*x + 48
sage: P.charpoly()
x^10 - 15*x^8 + 75*x^6 - 24*x^5 - 165*x^4 + 120*x^3 + 120*x^2 - 160*x + 48
sage: P.characteristic_polynomial(laplacian=True)
x^10 - 30*x^9 + 390*x^8 - 2880*x^7 + 13305*x^6 -
39882*x^5 + 77640*x^4 - 94800*x^3 + 66000*x^2 - 20000*x
```

charpoly(*var='x'*, *laplacian=False*)

Return the characteristic polynomial of the adjacency matrix of the (di)graph.

Let G be a (simple) graph with adjacency matrix A. Let I be the identity matrix of dimensions the same as A. The characteristic polynomial of G is defined as the determinant $\det(xI - A)$.

Note: `characteristic_polynomial` and `charpoly` are aliases and thus provide exactly the same method.

INPUT:

- `x` – (default: `'x'`); the variable of the characteristic polynomial
- `laplacian` – boolean (default: `False`); if `True`, use the Laplacian matrix

See also:

- *kirchhoff_matrix()*
- *laplacian_matrix()*

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.characteristic_polynomial()
x^10 - 15*x^8 + 75*x^6 - 24*x^5 - 165*x^4 + 120*x^3 + 120*x^2 - 160*x + 48
sage: P.charpoly()
x^10 - 15*x^8 + 75*x^6 - 24*x^5 - 165*x^4 + 120*x^3 + 120*x^2 - 160*x + 48
sage: P.characteristic_polynomial(laplacian=True)
x^10 - 30*x^9 + 390*x^8 - 2880*x^7 + 13305*x^6 -
39882*x^5 + 77640*x^4 - 94800*x^3 + 66000*x^2 - 20000*x
```

clear()

Empties the graph of vertices and edges and removes name, associated objects, and position information.

EXAMPLES:

```
sage: G=graphs.CycleGraph(4); G.set_vertices({0:'vertex0'})
sage: G.order(); G.size()
4
4
sage: len(G._pos)
4
sage: G.name()
'Cycle graph'
sage: G.get_vertex(0)
'vertex0'
sage: H = G.copy(sparse=True)
sage: H.clear()
```

(continues on next page)

(continued from previous page)

```
sage: H.order(); H.size()
0
0
sage: len(H._pos)
0
sage: H.name()
''
sage: H.get_vertex(0)
sage: H = G.copy(sparse=False)
sage: H.clear()
sage: H.order(); H.size()
0
0
sage: len(H._pos)
0
sage: H.name()
''
sage: H.get_vertex(0)
```

cluster_transitivity()

Return the transitivity (fraction of transitive triangles) of the graph.

Transitivity is the fraction of all existing triangles over all connected triples (triads), $T = 3 \times \frac{\text{triangles}}{\text{triads}}$.

See also section "Clustering" in chapter "Algorithms" of [HSS].

EXAMPLES:

```
sage: graphs.FruchtGraph().cluster_transitivity()
0.25
```

cluster_triangles(*nbunch=None*, *implementation=None*)

Return the number of triangles for the set $nbunch$ of vertices as a dictionary keyed by vertex.

See also section "Clustering" in chapter "Algorithms" of [HSS].

INPUT:

- `nbunch` – a list of vertices (default: `None`); the vertices to inspect. If ``nbunch=None``, returns data for all vertices in the graph.
- `implementation` – string (default: `None`); one of `'sparse_copy'`, `'dense_copy'`, `'networkx'` or `None` (default). In the latter case, the best algorithm available is used. Note that `'networkx'` does not support directed graphs.

EXAMPLES:

```
sage: F = graphs.FruchtGraph()
sage: list(F.cluster_triangles().values())
[1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0]
sage: F.cluster_triangles()
{0: 1, 1: 1, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 0, 9: 1, 10: 1, 11: 0}
sage: F.cluster_triangles(nbunch=[0, 1, 2])
{0: 1, 1: 1, 2: 0}
```

```
sage: G = graphs.RandomGNP(20, .3)
sage: d1 = G.cluster_triangles(implementation="networkx")
sage: d2 = G.cluster_triangles(implementation="dense_copy")
sage: d3 = G.cluster_triangles(implementation="sparse_copy")
```

(continues on next page)

(continued from previous page)

```
sage: d1 == d2 and d1 == d3
True
```

clustering_average(*implementation=None*)

Return the average clustering coefficient.

The clustering coefficient of a node i is the fraction of existing triangles containing node i over all possible triangles containing i: $c_i = T(i)/\binom{k_i}{2}$ where $T(i)$ is the number of existing triangles through i, and k_i is the degree of vertex i.

A coefficient for the whole graph is the average of the c_i.

See also section "Clustering" in chapter "Algorithms" of [HSS].

INPUT:

- `implementation` – string (default: `None`); one of `'boost'`, `'sparse_copy'`, `'dense_copy'`, `'networkx'` or `None` (default). In the latter case, the best algorithm available is used. Note that only `'networkx'` supports directed graphs.

EXAMPLES:

```
sage: (graphs.FruchtGraph()).clustering_average()
1/4
sage: (graphs.FruchtGraph()).clustering_average(implementation='networkx')
0.25
```

clustering_coeff(*nodes=None*, *weight=False*, *implementation=None*)

Return the clustering coefficient for each vertex in `nodes` as a dictionary keyed by vertex.

For an unweighted graph, the clustering coefficient of a node i is the fraction of existing triangles containing node i over all possible triangles containing i: $c_i = T(i)/\binom{k_i}{2}$ where $T(i)$ is the number of existing triangles through i, and k_i is the degree of vertex i.

For weighted graphs the clustering is defined as the geometric average of the subgraph edge weights, normalized by the maximum weight in the network.

The value of c_i is assigned 0 if $k_i < 2$.

See also section "Clustering" in chapter "Algorithms" of [HSS].

INPUT:

- `nodes` – an iterable container of vertices (default: `None`); the vertices to inspect. By default, returns data on all vertices in graph
- `weight` – string or boolean (default: `False`); if it is a string it uses the indicated edge property as weight. `weight = True` is equivalent to `weight = 'weight'`
- `implementation` – string (default: `None`); one of `'boost'`, `'sparse_copy'`, `'dense_copy'`, `'networkx'` or `None` (default). In the latter case, the best algorithm available is used. Note that only `'networkx'` supports directed or weighted graphs, and that `'sparse_copy'` and `'dense_copy'` do not support `node` different from `None`

EXAMPLES:

```
sage: graphs.FruchtGraph().clustering_coeff()
{0: 1/3, 1: 1/3, 2: 0, 3: 1/3, 4: 1/3, 5: 1/3,
 6: 1/3, 7: 1/3, 8: 0, 9: 1/3, 10: 1/3, 11: 0}

sage: (graphs.FruchtGraph()).clustering_coeff(weight=True)
```

(continues on next page)

(continued from previous page)

```
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0,
3: 0.3333333333333333, 4: 0.3333333333333333,
5: 0.3333333333333333, 6: 0.3333333333333333,
7: 0.3333333333333333, 8: 0, 9: 0.3333333333333333,
10: 0.3333333333333333, 11: 0}

sage: (graphs.FruchtGraph()).clustering_coeff(nodes=[0,1,2])
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.0}

sage: (graphs.FruchtGraph()).clustering_coeff(nodes=[0,1,2],
....:   weight=True)
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0}

sage: (graphs.GridGraph([5,5])).clustering_coeff(nodes=[(0,0),(0,1),(2,2)])
{(0, 0): 0.0, (0, 1): 0.0, (2, 2): 0.0}
```

coarsest_equitable_refinement(*partition*, *sparse=True*)

Return the coarsest partition which is finer than the input partition, and equitable with respect to self.

A partition is equitable with respect to a graph if for every pair of cells C_1, C_2 of the partition, the number of edges from a vertex of C_1 to C_2 is the same, over all vertices in C_1.

A partition P_1 is finer than P_2 (P_2 is coarser than P_1) if every cell of P_1 is a subset of a cell of P_2.

INPUT:

- `partition` – a list of lists
- **`sparse` – boolean (default: `False`); whether to use sparse or** dense representation - for small graphs, use dense for speed

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.coarsest_equitable_refinement([[0],list(range(1,10))])
[[0], [2, 3, 6, 7, 8, 9], [1, 4, 5]]
sage: G = graphs.CubeGraph(3)
sage: verts = G.vertices()
sage: Pi = [verts[:1], verts[1:]]
sage: Pi
[['000'], ['001', '010', '011', '100', '101', '110', '111']]
sage: [sorted(cell) for cell in G.coarsest_equitable_refinement(Pi)]
[['000'], ['011', '101', '110'], ['111'], ['001', '010', '100']]
```

Note that given an equitable partition, this function returns that partition:

```
sage: P = graphs.PetersenGraph()
sage: prt = [[0], [1, 4, 5], [2, 3, 6, 7, 8, 9]]
sage: P.coarsest_equitable_refinement(prt)
[[0], [1, 4, 5], [2, 3, 6, 7, 8, 9]]
```

```
sage: ss = (graphs.WheelGraph(6)).line_graph(labels=False)
sage: prt = [[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3,␣
↪4)]]
sage: ss.coarsest_equitable_refinement(prt)
Traceback (most recent call last):
...
TypeError: partition ([[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)],␣
↪[(2, 3), (3, 4)]]) is not valid for this graph: vertices are incorrect
```

(continues on next page)

(continued from previous page)

```
sage: ss = (graphs.WheelGraph(5)).line_graph(labels=False)
sage: ss.coarsest_equitable_refinement(prt)
[[(0, 1)], [(1, 2), (1, 4)], [(0, 3)], [(0, 4), (0, 2)], [(2, 3), (3, 4)]]
```

ALGORITHM: Brendan D. McKay's Master's Thesis, University of Melbourne, 1976.

complement()

Return the complement of the (di)graph.

The complement of a graph has the same vertices, but exactly those edges that are not in the original graph. This is not well defined for graphs with multiple edges.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.plot() # long time
Graphics object consisting of 26 graphics primitives
sage: PC = P.complement()
sage: PC.plot() # long time
Graphics object consisting of 41 graphics primitives
```

```
sage: graphs.TetrahedralGraph().complement().size()
0
sage: graphs.CycleGraph(4).complement().edges()
[(0, 2, None), (1, 3, None)]
sage: graphs.CycleGraph(4).complement()
complement(Cycle graph): Graph on 4 vertices
sage: G = Graph(multiedges=True, sparse=True)
sage: G.add_edges([(0, 1)] * 3)
sage: G.complement()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with
multiedges. Perhaps this method can be updated to handle them, but
in the meantime if you want to use it please disallow multiedges
using allow_multiple_edges().
```

connected_component_containing_vertex(*G*, *vertex*, *sort=True*)

Return a list of the vertices connected to vertex.

INPUT:

- `G` – the input graph
- `v` – the vertex to search for
- `sort` – boolean (default `True`); whether to sort vertices inside the component

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_component_containing_
↪vertex
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_component_containing_vertex(G, 0)
[0, 1, 2, 3]
sage: G.connected_component_containing_vertex(0)
[0, 1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_component_containing_vertex(D, 0)
[0, 1, 2, 3]
```

connected_components(*G*, *sort=True*)

Return the list of connected components.

This returns a list of lists of vertices, each list representing a connected component. The list is ordered from largest to smallest component.

INPUT:

- `G` – the input graph
- `sort` – boolean (default `True`); whether to sort vertices inside each component

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components(G)
[[0, 1, 2, 3], [4, 5, 6]]
sage: G.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components(D)
[[0, 1, 2, 3], [4, 5, 6]]
```

connected_components_number(*G*)

Return the number of connected components.

INPUT:

- `G` – the input graph

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_number
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components_number(G)
2
sage: G.connected_components_number()
2
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components_number(D)
2
```

connected_components_sizes(*G*)

Return the sizes of the connected components as a list.

The list is sorted from largest to lower values.

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_sizes
sage: for x in graphs(3):
....:     print(connected_components_sizes(x))
[1, 1, 1]
[2, 1]
[3]
```

(continues on next page)

(continued from previous page)

```
[3]
sage: for x in graphs(3):
....:     print(x.connected_components_sizes())
[1, 1, 1]
[2, 1]
[3]
[3]
```

connected_components_subgraphs(*G*)

Return a list of connected components as graph objects.

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_subgraphs
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: L = connected_components_subgraphs(G)
sage: graphs_list.show_graphs(L)
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: L = connected_components_subgraphs(D)
sage: graphs_list.show_graphs(L)
sage: L = D.connected_components_subgraphs()
sage: graphs_list.show_graphs(L)
```

connected_subgraph_iterator(*G*, *k=None*, *vertices_only=False*)

Iterator over the induced connected subgraphs of order at most k.

This method implements a iterator over the induced connected subgraphs of the input (di)graph. An induced subgraph of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset (Wikipedia article Induced_subgraph).

As for method `sage.graphs.generic_graph.connected_components()`, edge orientation is ignored. Hence, the directed graph with a single arc $0 \to 1$ is considered connected.

INPUT:

- `G` – a *Graph* or a *DiGraph*; loops and multiple edges are allowed
- `k` – (optional) integer; maximum order of the connected subgraphs to report; by default, the method iterates over all connected subgraphs (equivalent to `k == n`)
- `vertices_only` – boolean (default: `False`); whether to return (Di)Graph or list of vertices

EXAMPLES:

```
sage: G = DiGraph([(1, 2), (2, 3), (3, 4), (4, 2)])
sage: list(G.connected_subgraph_iterator())
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 4 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(vertices_only=True))
```

(continues on next page)

(continued from previous page)

```
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 4],
 [2], [2, 3], [2, 3, 4], [2, 4], [3], [3, 4], [4]]
sage: list(G.connected_subgraph_iterator(k=2))
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(k=2, vertices_only=True))
[[1], [1, 2], [2], [2, 3], [2, 4], [3], [3, 4], [4]]

sage: G = DiGraph([(1, 2), (2, 1)])
sage: list(G.connected_subgraph_iterator())
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(vertices_only=True))
[[1], [1, 2], [2]]
```

contract_edge(*u*, *v=None*, *label=None*)

Contract an edge from `u` to `v`.

This method returns silently if the edge does not exist.

INPUT: The following forms are all accepted:

- G.contract_edge(1, 2)
- G.contract_edge((1, 2))
- G.contract_edge([(1, 2)])
- G.contract_edge(1, 2, 'label')
- G.contract_edge((1, 2, 'label'))
- G.contract_edge([(1, 2, 'label')])

EXAMPLES:

```
sage: G = graphs.CompleteGraph(4)
sage: G.contract_edge((0, 1)); G.edges()
[(0, 2, None), (0, 3, None), (2, 3, None)]
sage: G = graphs.CompleteGraph(4)
sage: G.allow_loops(True); G.allow_multiple_edges(True)
sage: G.contract_edge((0, 1)); G.edges()
[(0, 2, None), (0, 2, None), (0, 3, None), (0, 3, None), (2, 3, None)]
sage: G.contract_edge((0, 2)); G.edges()
[(0, 0, None), (0, 3, None), (0, 3, None), (0, 3, None)]
```

```
sage: G = graphs.CompleteGraph(4).to_directed()
sage: G.allow_loops(True)
sage: G.contract_edge(0, 1); G.edges()
[(0, 0, None),
 (0, 2, None),
 (0, 3, None),
```

(continues on next page)

(continued from previous page)

```
 (2, 0, None),
 (2, 3, None),
 (3, 0, None),
 (3, 2, None)]
```

contract_edges(*edges*)

Contract edges from an iterable container.

If e is an edge that is not contracted but the vertices of e are merged by contraction of other edges, then e will become a loop.

INPUT:

- `edges` – a list containing 2-tuples or 3-tuples that represent edges

EXAMPLES:

```
sage: G = graphs.CompleteGraph(4)
sage: G.allow_loops(True); G.allow_multiple_edges(True)
sage: G.contract_edges([(0, 1), (1, 2), (0, 2)]); G.edges()
[(0, 3, None), (0, 3, None), (0, 3, None)]
sage: G.contract_edges([(1, 3), (2, 3)]); G.edges()
[(0, 3, None), (0, 3, None), (0, 3, None)]
sage: G = graphs.CompleteGraph(4)
sage: G.allow_loops(True); G.allow_multiple_edges(True)
sage: G.contract_edges([(0, 1), (1, 2), (0, 2), (1, 3), (2, 3)]); G.edges()
[(0, 0, None)]
```

```
sage: D = digraphs.Complete(4)
sage: D.allow_loops(True); D.allow_multiple_edges(True)
sage: D.contract_edges([(0, 1), (1, 0), (0, 2)]); D.edges()
[(0, 0, None),
 (0, 0, None),
 (0, 0, None),
 (0, 3, None),
 (0, 3, None),
 (0, 3, None),
 (3, 0, None),
 (3, 0, None),
 (3, 0, None)]
```

copy(*weighted=None*, *data_structure=None*, *sparse=None*, *immutable=None*)

Change the graph implementation

INPUT:

- `weighted` – boolean (default: `None`); weightedness for the copy. Might change the equality class if not `None`.
- `sparse` – boolean (default: `None`); `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`. Only used when `data_structure=None`.
- `data_structure` – string (default: `None`); one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of *Graph* or *DiGraph*.
- `immutable` – boolean (default: `None`); whether to create a mutable/immutable copy. Only used when `data_structure=None`.
  - `immutable=None` (default) means that the graph and its copy will behave the same way.

- `immutable=True` is a shortcut for `data_structure='static_sparse'`
- `immutable=False` means that the created graph is mutable. When used to copy an immutable graph, the data structure used is `"sparse"` unless anything else is specified.

Note: If the graph uses *StaticSparseBackend* and the `_immutable` flag, then `self` is returned rather than a copy (unless one of the optional arguments is used).

OUTPUT:

A Graph object.

Warning: Please use this method only if you need to copy but change the underlying data structure or weightedness. Otherwise simply do `copy(g)` instead of `g.copy()`.

Warning: If `weighted` is passed and is not the weightedness of the original, then the copy will not equal the original.

EXAMPLES:

```
sage: g = Graph({0: [0, 1, 1, 2]}, loops=True, multiedges=True, sparse=True)
sage: g == copy(g)
True
sage: g = DiGraph({0: [0, 1, 1, 2], 1: [0, 1]}, loops=True, multiedges=True,␣
↪sparse=True)
sage: g == copy(g)
True
```

Note that vertex associations are also kept:

```
sage: d = {0: graphs.DodecahedralGraph(), 1: graphs.FlowerSnark(), 2: graphs.
↪MoebiusKantorGraph(), 3: graphs.PetersenGraph()}
sage: T = graphs.TetrahedralGraph()
sage: T.set_vertices(d)
sage: T2 = copy(T)
sage: T2.get_vertex(0)
Dodecahedron: Graph on 20 vertices
```

Notice that the copy is at least as deep as the objects:

```
sage: T2.get_vertex(0) is T.get_vertex(0)
False
```

Examples of the keywords in use:

```
sage: G = graphs.CompleteGraph(9)
sage: H = G.copy()
sage: H == G; H is G
True
False
sage: G1 = G.copy(sparse=True)
sage: G1 == G
True
```

(continues on next page)

(continued from previous page)

```
sage: G1 is G
False
sage: G2 = copy(G)
sage: G2 is G
False
```

Argument `weighted` affects the equality class:

```
sage: G = graphs.CompleteGraph(5)
sage: H1 = G.copy(weighted=False)
sage: H2 = G.copy(weighted=True)
sage: [G.weighted(), H1.weighted(), H2.weighted()]
[False, False, True]
sage: [G == H1, G == H2, H1 == H2]
[True, False, False]
sage: G.weighted(True)
sage: [G == H1, G == H2, H1 == H2]
[False, True, False]
```

crossing_number()

Return the crossing number of the graph.

The crossing number of a graph is the minimum number of edge crossings needed to draw the graph on a plane. It can be seen as a measure of non-planarity; a planar graph has crossing number zero.

See the Wikipedia article Crossing_number for more information.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.crossing_number()
2
```

ALGORITHM:

This is slow brute force implementation: for every k pairs of edges try adding a new vertex for a crossing point for them. If the result is not planar in any of those, try $k + 1$ pairs.

Computing the crossing number is NP-hard problem.

cycle_basis(*output='vertex'*)

Return a list of cycles which form a basis of the cycle space of `self`.

A basis of cycles of a graph is a minimal collection of cycles (considered as sets of edges) such that the edge set of any cycle in the graph can be written as a $Z/2Z$ sum of the cycles in the basis.

See the Wikipedia article Cycle_basis for more information.

INPUT:

- `output` – string (default: `'vertex'`); whether every cycle is given as a list of vertices (`output == 'vertex'`) or a list of edges (`output == 'edge'`)

OUTPUT:

A list of lists, each of them representing the vertices (or the edges) of a cycle in a basis.

ALGORITHM:

Uses the NetworkX library for graphs without multiple edges.

Otherwise, by the standard algorithm using a spanning tree.

EXAMPLES:

A cycle basis in Petersen's Graph

```
sage: g = graphs.PetersenGraph()
sage: g.cycle_basis()  # py2
[[1, 2, 7, 5, 0], [8, 3, 2, 7, 5], [4, 3, 2, 7, 5, 0], [4, 9, 7, 5, 0], [8, 6,
↪ 9, 7, 5], [1, 6, 9, 7, 5, 0]]
sage: g.cycle_basis()  # py3
[[1, 6, 8, 5, 0], [4, 9, 6, 8, 5, 0], [7, 9, 6, 8, 5], [4, 3, 8, 5, 0], [1, 2,
↪ 3, 8, 5, 0], [7, 2, 3, 8, 5]]
```

One can also get the result as a list of lists of edges:

```
sage: g.cycle_basis(output='edge')  # py2
[[(1, 2, None), (2, 7, None), (7, 5, None), (5, 0, None),
(0, 1, None)], [(8, 3, None), (3, 2, None), (2, 7, None),
(7, 5, None), (5, 8, None)], [(4, 3, None), (3, 2, None),
(2, 7, None), (7, 5, None), (5, 0, None), (0, 4, None)],
[(4, 9, None), (9, 7, None), (7, 5, None), (5, 0, None),
(0, 4, None)], [(8, 6, None), (6, 9, None), (9, 7, None),
(7, 5, None), (5, 8, None)], [(1, 6, None), (6, 9, None),
(9, 7, None), (7, 5, None), (5, 0, None), (0, 1, None)]]
sage: g.cycle_basis(output='edge')  # py3
[[(1, 6, None), (6, 8, None), (8, 5, None), (5, 0, None),
(0, 1, None)], [(4, 9, None), (9, 6, None), (6, 8, None),
(8, 5, None), (5, 0, None), (0, 4, None)], [(7, 9, None),
(9, 6, None), (6, 8, None), (8, 5, None), (5, 7, None)],
[(4, 3, None), (3, 8, None), (8, 5, None), (5, 0, None),
(0, 4, None)], [(1, 2, None), (2, 3, None), (3, 8, None),
(8, 5, None), (5, 0, None), (0, 1, None)], [(7, 2, None),
(2, 3, None), (3, 8, None), (8, 5, None), (5, 7, None)]]
```

Checking the given cycles are algebraically free:

```
sage: g = graphs.RandomGNP(30, .4)
sage: basis = g.cycle_basis()
```

Building the space of (directed) edges over $Z/2Z$. On the way, building a dictionary associating a unique vector to each undirected edge:

```
sage: m = g.size()
sage: edge_space = VectorSpace(FiniteField(2), m)
sage: edge_vector = dict(zip(g.edges(labels=False, sort=False), edge_space.
↪basis()))
sage: for (u, v), vec in list(edge_vector.items()):
....:     edge_vector[(v, u)] = vec
```

Defining a lambda function associating a vector to the vertices of a cycle:

```
sage: vertices_to_edges = lambda x: zip(x, x[1:] + [x[0]])
sage: cycle_to_vector = lambda x: sum(edge_vector[e] for e in vertices_to_
↪edges(x))
```

Finally checking the cycles are a free set:

```
sage: basis_as_vectors = [cycle_to_vector(_) for _ in basis]
sage: edge_space.span(basis_as_vectors).rank() == len(basis)
True
```

For undirected graphs with multiple edges:

```
sage: G = Graph([(0, 2, 'a'), (0, 2, 'b'), (0, 1, 'c'), (1, 2, 'd')],␣
↪multiedges=True)
sage: G.cycle_basis()
[[0, 2], [2, 1, 0]]
sage: G.cycle_basis(output='edge')
[[(0, 2, 'a'), (2, 0, 'b')], [(2, 1, 'd'), (1, 0, 'c'), (0, 2, 'a')]]
sage: H = Graph([(1, 2), (2, 3), (2, 3), (3, 4), (1, 4), (1, 4), (4, 5), (5,␣
↪6), (4, 6), (6, 7)], multiedges=True)
sage: H.cycle_basis()
[[1, 4], [2, 3], [4, 3, 2, 1], [6, 5, 4]]
```

Disconnected graph:

```
sage: G.add_cycle(["Hey", "Wuuhuu", "Really ?"])
sage: [sorted(c) for c in G.cycle_basis()]
[['Hey', 'Really ?', 'Wuuhuu'], [0, 2], [0, 1, 2]]
sage: [sorted(c) for c in G.cycle_basis(output='edge')]  # py2
[[('Hey', 'Wuuhuu', None),
  ('Really ?', 'Hey', None),
  ('Wuuhuu', 'Really ?', None)],
 [(0, 2, 'a'), (2, 0, 'b')],
 [(0, 2, 'b'), (1, 0, 'c'), (2, 1, 'd')]]
sage: [sorted(c) for c in G.cycle_basis(output='edge')]  # py3
[[('Hey', 'Really ?', None),
  ('Really ?', 'Wuuhuu', None),
  ('Wuuhuu', 'Hey', None)],
 [(0, 2, 'a'), (2, 0, 'b')],
 [(0, 2, 'b'), (1, 0, 'c'), (2, 1, 'd')]]
```

Graph that allows multiple edges but does not contain any:

```
sage: G = graphs.CycleGraph(3)
sage: G.allow_multiple_edges(True)
sage: G.cycle_basis()
[[2, 1, 0]]
```

Not yet implemented for directed graphs:

```
sage: G = DiGraph([(0, 2, 'a'), (0, 1, 'c'), (1, 2, 'd')])
sage: G.cycle_basis()
Traceback (most recent call last):
...
NotImplementedError: not implemented for directed graphs
```

degree (*vertices=None*, *labels=False*)

Return the degree (in + out for digraphs) of a vertex or of vertices.

INPUT:

- `vertices` – a vertex or an iterable container of vertices (default: `None`); if `vertices` is a single vertex, returns the number of neighbors of that vertex. If `vertices` is an iterable container of vertices, returns a list of degrees. If `vertices` is `None`, same as listing all vertices.
- `labels` – boolean (default: `False`); when `True`, return a dictionary mapping each vertex in `vertices` to its degree. Otherwise, return the degree of a single vertex or a list of the degrees of each vertex in `vertices`

OUTPUT:

- When `vertices` is a single vertex and `labels` is `False`, returns the degree of that vertex as an integer
- When `vertices` is an iterable container of vertices (or `None`) and `labels` is `False`, returns a list of integers. The i-th value is the degree of the i-th vertex in the list `vertices`. When `vertices` is `None`, the i-th value is the degree of i-th vertex in the ordering `list(self)`, which might be different from the ordering of the vertices given by `g.vertices()`.
- When `labels` is `True`, returns a dictionary mapping each vertex in `vertices` to its degree

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.degree(5)
3
```

```
sage: K = graphs.CompleteGraph(9)
sage: K.degree()
[8, 8, 8, 8, 8, 8, 8, 8, 8]
```

```
sage: D = DiGraph({0: [1, 2, 3], 1: [0, 2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.degree(vertices=[0, 1, 2], labels=True)
{0: 5, 1: 4, 2: 3}
sage: D.degree()
[5, 4, 3, 3, 3, 2]
```

When `vertices=None` and `labels=False`, the i-th value of the returned list is the degree of the i-th vertex in the list `list(self)`:

```
sage: D = digraphs.DeBruijn(4, 2)
sage: D.delete_vertex('20')
sage: print(D.degree())                          # py2
[6, 7, 7, 7, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8, 8]
sage: print(D.degree(vertices=list(D)))          # py2
[6, 7, 7, 7, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8, 8]
sage: print(D.degree(vertices=D.vertices()))     # py2
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
sage: print(D.degree())                          # py3
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
sage: print(D.degree(vertices=list(D)))          # py3
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
sage: print(D.degree(vertices=D.vertices()))     # py3
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
```

degree_histogram()

Return a list, whose i-th entry is the frequency of degree i.

EXAMPLES:

```
sage: G = graphs.Grid2dGraph(9, 12)
sage: G.degree_histogram()
[0, 0, 4, 34, 70]
```

```
sage: G = graphs.Grid2dGraph(9, 12).to_directed()
sage: G.degree_histogram()
[0, 0, 0, 0, 4, 0, 34, 0, 70]
```

degree_iterator(*vertices=None*, *labels=False*)

Return an iterator over the degrees of the (di)graph.

In the case of a digraph, the degree is defined as the sum of the in-degree and the out-degree, i.e. the total number of edges incident to a given vertex.

INPUT:

- `vertices` – a vertex or an iterable container of vertices (default: `None`); if `vertices` is a single vertex, the iterator will yield the number of neighbors of that vertex. If `vertices` is an iterable container of vertices, return an iterator over the degrees of these vertices. If `vertices` is `None`, same as listing all vertices.
- `labels` – boolean (default: `False`); whether to return an iterator over degrees (`labels=False`), or over tuples `(vertex, degree)`

Note: The returned iterator yields values in order specified by `list(vertices)`. When `vertices` is `None`, it yields values in the same order as `list(self)`, which might be different from the ordering of the vertices given by `g.vertices()`.

EXAMPLES:

```
sage: G = graphs.Grid2dGraph(3, 4)
sage: for i in G.degree_iterator():  # py2
....:     print(i)                   # py2
3
4
2
...
2
4
sage: for i in G.degree_iterator():  # py3
....:     print(i)                   # py3
2
3
3
...
3
2
sage: for i in G.degree_iterator(labels=True):  # py2
....:  print(i)                                 # py2
((0, 1), 3)
((1, 2), 4)
((0, 0), 2)
...
((0, 3), 2)
((1, 1), 4)
sage: for i in G.degree_iterator(labels=True):  # py3
....:  print(i)                                 # py3
((0, 0), 2)
((0, 1), 3)
((0, 2), 3)
...
((2, 2), 3)
((2, 3), 2)
```

```
sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: for i in D.degree_iterator():              # py2
....:     print(i)                               # py2
6
```

(continues on next page)

(continued from previous page)

```
6
...
4
6
sage: for i in D.degree_iterator():                # py3
....:     print(i)                                 # py3
4
6
...
6
4
sage: for i in D.degree_iterator(labels=True):     # py2
....:     print(i)                                 # py2
((0, 1), 6)
((1, 2), 6)
...
((1, 0), 4)
((0, 2), 6)
sage: for i in D.degree_iterator(labels=True):     # py3
....:     print(i)                                 # py3
((0, 0), 4)
((0, 1), 6)
...
((1, 2), 6)
((1, 3), 4)
```

When `vertices=None` yields values in the order of `list(D)`:

```
sage: V = list(D)
sage: D = digraphs.DeBruijn(4, 2)
sage: D.delete_vertex('20')
sage: print(list(D.degree_iterator()))   # py2
[6, 7, 7, 7, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8, 8]
sage: print([D.degree(v) for v in D])    # py2
[6, 7, 7, 7, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8, 8]
sage: print(list(D.degree_iterator()))   # py3
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
sage: print([D.degree(v) for v in D])    # py3
[7, 7, 6, 7, 8, 8, 7, 8, 8, 7, 8, 8, 8, 7, 8]
```

degree_sequence()

Return the degree sequence of this (di)graph.

EXAMPLES:

The degree sequence of an undirected graph:

```
sage: g = Graph({1: [2, 5], 2: [1, 5, 3, 4], 3: [2, 5], 4: [3], 5: [2, 3]})
sage: g.degree_sequence()
[4, 3, 3, 2, 2]
```

The degree sequence of a digraph:

```
sage: g = DiGraph({1: [2, 5, 6], 2: [3, 6], 3: [4, 6], 4: [6], 5: [4, 6]})
sage: g.degree_sequence()
[5, 3, 3, 3, 3, 3]
```

Degree sequences of some common graphs:

```
sage: graphs.PetersenGraph().degree_sequence()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
sage: graphs.HouseGraph().degree_sequence()
[3, 3, 2, 2, 2]
sage: graphs.FlowerSnark().degree_sequence()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

degree_to_cell(*vertex*, *cell*)

Returns the number of edges from vertex to an edge in cell. In the case of a digraph, returns a tuple (in_degree, out_degree).

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: cell = G.vertices()[:3]
sage: G.degree_to_cell('011', cell)
2
sage: G.degree_to_cell('111', cell)
0
```

```
sage: D = DiGraph({ 0:[1,2,3], 1:[3,4], 3:[4,5]})
sage: cell = [0,1,2]
sage: D.degree_to_cell(5, cell)
(0, 0)
sage: D.degree_to_cell(3, cell)
(2, 0)
sage: D.degree_to_cell(0, cell)
(0, 2)
```

delete_edge(*u*, *v=None*, *label=None*)

Delete the edge from `u` to `v`.

This method returns silently if vertices or edge does not exist.

INPUT: The following forms are all accepted:

- G.delete_edge(1, 2)
- G.delete_edge((1, 2))
- G.delete_edges([(1, 2)])
- G.delete_edge(1, 2, 'label')
- G.delete_edge((1, 2, 'label'))
- G.delete_edges([(1, 2, 'label')])

EXAMPLES:

```
sage: G = graphs.CompleteGraph(9)
sage: G.size()
36
sage: G.delete_edge( 1, 2 )
sage: G.delete_edge( (3, 4) )
sage: G.delete_edges( [ (5, 6), (7, 8) ] )
sage: G.size()
32
```

```
sage: G.delete_edge( 2, 3, 'label' )
sage: G.delete_edge( (4, 5, 'label') )
sage: G.delete_edges( [ (6, 7, 'label') ] )
sage: G.size()
32
sage: G.has_edge( (4, 5) ) # correct!
True
sage: G.has_edge( (4, 5, 'label') ) # correct!
False
```

```
sage: C = digraphs.Complete(9)
sage: C.size()
72
sage: C.delete_edge( 1, 2 )
sage: C.delete_edge( (3, 4) )
sage: C.delete_edges( [ (5, 6), (7, 8) ] )
sage: C.size()
68
```

```
sage: C.delete_edge( 2, 3, 'label' )
sage: C.delete_edge( (4, 5, 'label') )
sage: C.delete_edges( [ (6, 7, 'label') ] )
sage: C.size() # correct!
68
sage: C.has_edge( (4, 5) ) # correct!
True
sage: C.has_edge( (4, 5, 'label') ) # correct!
False
```

delete_edges(*edges*)

Delete edges from an iterable container.

EXAMPLES:

```
sage: K12 = graphs.CompleteGraph(12)
sage: K4 = graphs.CompleteGraph(4)
sage: K12.size()
66
sage: K12.delete_edges(K4.edge_iterator())
sage: K12.size()
60
```

```
sage: K12 = digraphs.Complete(12)
sage: K4 = digraphs.Complete(4)
sage: K12.size()
132
sage: K12.delete_edges(K4.edge_iterator())
sage: K12.size()
120
```

delete_multiedge(*u*, *v*)

Delete all edges from `u` to `v`.

EXAMPLES:

```
sage: G = Graph(multiedges=True, sparse=True)
sage: G.add_edges([(0, 1), (0, 1), (0, 1), (1, 2), (2, 3)])
```

(continues on next page)

(continued from previous page)

```
sage: G.edges()
[(0, 1, None), (0, 1, None), (0, 1, None), (1, 2, None), (2, 3, None)]
sage: G.delete_multiedge(0, 1)
sage: G.edges()
[(1, 2, None), (2, 3, None)]
```

```
sage: D = DiGraph(multiedges=True, sparse=True)
sage: D.add_edges([(0, 1, 1), (0, 1, 2), (0, 1, 3), (1, 0, None), (1, 2,␣
↪None), (2, 3, None)])
sage: D.edges()
[(0, 1, 1), (0, 1, 2), (0, 1, 3), (1, 0, None), (1, 2, None), (2, 3, None)]
sage: D.delete_multiedge(0, 1)
sage: D.edges()
[(1, 0, None), (1, 2, None), (2, 3, None)]
```

delete_vertex(*vertex*, *in_order=False*)

Delete vertex, removing all incident edges.

Deleting a non-existent vertex will raise an exception.

INPUT:

- `in_order` – boolean (default: `False`); if `True`, this deletes the i-th vertex in the sorted list of vertices, i.e. `G.vertices()[i]`

EXAMPLES:

```
sage: G = Graph(graphs.WheelGraph(9))
sage: G.delete_vertex(0); G.show()
```

```
sage: D = DiGraph({0: [1, 2, 3, 4, 5], 1: [2], 2: [3], 3: [4], 4: [5], 5: [1]}
↪)
sage: D.delete_vertex(0); D
Digraph on 5 vertices
sage: D.vertices()
[1, 2, 3, 4, 5]
sage: D.delete_vertex(0)
Traceback (most recent call last):
...
ValueError: vertex (0) not in the graph
```

```
sage: G = graphs.CompleteGraph(4).line_graph(labels=False)
sage: G.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G.delete_vertex(0, in_order=True)
sage: G.vertices()
[(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G = graphs.PathGraph(5)
sage: G.set_vertices({0: 'no delete', 1: 'delete'})
sage: G.delete_vertex(1)
sage: G.get_vertices()
{0: 'no delete', 2: None, 3: None, 4: None}
sage: G.get_pos()
{0: (0, 0), 2: (2, 0), 3: (3, 0), 4: (4, 0)}
```

delete_vertices(*vertices*)

Delete vertices from the (di)graph taken from an iterable container of vertices.

Deleting a non-existent vertex will raise an exception, in which case none of the vertices in `vertices` is deleted.

EXAMPLES:

```
sage: D = DiGraph({0: [1, 2, 3, 4, 5], 1: [2], 2: [3], 3: [4], 4: [5], 5: [1]}
↪)
sage: D.delete_vertices([1, 2, 3, 4, 5]); D
Digraph on 1 vertex
sage: D.vertices()
[0]
sage: D.delete_vertices([1])
Traceback (most recent call last):
...
ValueError: vertex (1) not in the graph
```

density()

Return the density of the (di)graph.

The density of a (di)graph is defined as the number of edges divided by number of possible edges.

In the case of a multigraph, raises an error, since there is an infinite number of possible edges.

EXAMPLES:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7, 8], 6: [8,
↪9], 7: [9]}
sage: G = Graph(d); G.density()
1/3
sage: G = Graph({0: [1, 2], 1: [0]}); G.density()
2/3
sage: G = DiGraph({0: [1, 2], 1: [0]}); G.density()
1/2
```

Note that there are more possible edges on a looped graph:

```
sage: G.allow_loops(True)
sage: G.density()
1/3
```

depth_first_search(*start*, *ignore_direction=False*, *neighbors=None*, *edges=False*)

Return an iterator over the vertices in a depth-first ordering.

INPUT:

- `start` – vertex or list of vertices from which to start the traversal
- `ignore_direction` – boolean (default `False`); only applies to directed graphs. If `True`, searches across edges in either direction.
- `neighbors` – function (default: `None`); a function that inputs a vertex and return a list of vertices. For an undirected graph, `neighbors` is by default the *neighbors()* function. For a digraph, the `neighbors` function defaults to the *neighbor_out_iterator()* function of the graph.
- `edges` – boolean (default: `False`); whether to return the edges of the DFS tree in the order of visit or the vertices (default). Edges are directed in root to leaf orientation of the tree.

See also:

- *breadth_first_search()*
- *breadth_first_search* – breadth-first search for fast compiled graphs.

- *depth_first_search* – depth-first search for fast compiled graphs.

EXAMPLES:

```
sage: G = Graph({0: [1], 1: [2], 2: [3], 3: [4], 4: [0]})
sage: list(G.depth_first_search(0))
[0, 4, 3, 2, 1]
```

By default, the edge direction of a digraph is respected, but this can be overridden by the `ignore_direction` parameter:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.depth_first_search(0))
[0, 3, 6, 7, 2, 5, 1, 4]
sage: list(D.depth_first_search(0, ignore_direction=True))
[0, 7, 6, 3, 5, 2, 1, 4]
```

Multiple starting vertices can be specified in a list:

```
sage: D = DiGraph({0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [6], 5: [7], 6: [7],␣
↪7: [0]})
sage: list(D.depth_first_search([0]))
[0, 3, 6, 7, 2, 5, 1, 4]
sage: list(D.depth_first_search([0, 6]))
[0, 3, 6, 7, 2, 5, 1, 4]
```

More generally, you can specify a `neighbors` function. For example, you can traverse the graph backwards by setting `neighbors` to be the *neighbors_in()* function of the graph:

```
sage: D = digraphs.Path(10)
sage: D.add_path([22, 23, 24, 5])
sage: D.add_path([5, 33, 34, 35])
sage: list(D.depth_first_search(5, neighbors=D.neighbors_in))
[5, 4, 3, 2, 1, 0, 24, 23, 22]
sage: list(D.breadth_first_search(5, neighbors=D.neighbors_in))
[5, 24, 4, 23, 3, 22, 2, 1, 0]
sage: list(D.depth_first_search(5, neighbors=D.neighbors_out))
[5, 6, 7, 8, 9, 33, 34, 35]
sage: list(D.breadth_first_search(5, neighbors=D.neighbors_out))
[5, 33, 6, 34, 7, 35, 8, 9]
```

You can get edges of the DFS tree instead of the vertices using the `edges` parameter:

```
sage: D = DiGraph({1: [2, 3], 2: [4], 3: [4], 4: [1, 5], 5: [2, 6]})
sage: list(D.depth_first_search(1, edges=True))
[(1, 3), (3, 4), (4, 5), (5, 6), (5, 2)]
sage: list(D.depth_first_search(1, ignore_direction=True, edges=True))
[(1, 4), (4, 5), (5, 6), (5, 2), (4, 3)]
```

diameter(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the diameter of the (di)graph.

The diameter is defined to be the maximum distance between two vertices. It is infinite if the (di)graph is not (strongly) connected.

For more information and examples on how to use input variables, see *shortest_paths()* and *eccentricity()*

INPUT:

- by_weight – boolean (default: False); if True, edge weights are taken into account; if False, all edges have weight 1
- algorithm – string (default: None); one of the following algorithms:
  - 'BFS': the computation is done through a BFS centered on each vertex successively. Works only if by_weight==False.
  - 'Floyd-Warshall-Cython': a Cython implementation of the Floyd-Warshall algorithm. Works only if by_weight==False and v is None.
  - 'Floyd-Warshall-Python': a Python implementation of the Floyd-Warshall algorithm. Works also with weighted graphs, even with negative weights (but no negative cycle is allowed). However, v must be None.
  - 'Dijkstra_NetworkX': the Dijkstra algorithm, implemented in NetworkX. It works with weighted graphs, but no negative weight is allowed.
  - 'standard', '2sweep',“‘2Dsweep’, ``'multi-sweep', 'iFUB': these algorithms are implemented in *sage.graphs.distances_all_pairs.diameter()* They work only if by_weight==False. See the function documentation for more information.
  - 'Dijkstra_Boost': the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - 'Johnson_Boost': the Johnson algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - None (default): Sage chooses the best algorithm: 'iFUB' for unweighted graphs, 'Dijkstra_Boost' if all weights are positive, 'Johnson_Boost' otherwise.
- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

The more symmetric a graph is, the smaller (diameter - radius) is:

```
sage: G = graphs.BarbellGraph(9, 3)
sage: G.radius()
3
sage: G.diameter()
6
```

```
sage: G = graphs.OctahedralGraph()
sage: G.radius()
2
sage: G.diameter()
2
```

disjoint_routed_paths(*pairs*, *solver=None*, *verbose=0*)

Return a set of disjoint routed paths.

Given a set of pairs (s_i, t_i), a set of disjoint routed paths is a set of $s_i - t_i$ paths which can intersect at their endpoints and are vertex-disjoint otherwise.

INPUT:

- pairs – list of pairs of vertices
- solver – string (default: None); specifies a Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default (quiet).

EXAMPLES:

Given a grid, finding two vertex-disjoint paths, the first one from the top-left corner to the bottom-left corner, and the second from the top-right corner to the bottom-right corner is easy:

```
sage: g = graphs.Grid2dGraph(5, 5)
sage: p1,p2 = g.disjoint_routed_paths([((0, 0), (0, 4)), ((4, 4), (4, 0))])
```

Though there is obviously no solution to the problem in which each corner is sending information to the opposite one:

```
sage: g = graphs.Grid2dGraph(5, 5)
sage: p1,p2 = g.disjoint_routed_paths([((0, 0), (4, 4)), ((0, 4), (4, 0))])
Traceback (most recent call last):
...
EmptySetError: the disjoint routed paths do not exist
```

disjoint_union(*other*, *labels='pairs'*, *immutable=None*)

Return the disjoint union of self and other.

INPUT:

- labels – string (default: 'pairs'); if set to 'pairs', each element v in the first graph will be named (0, v) and each element u in other will be named (1, u) in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.
- immutable – boolean (default: None); whether to create a mutable/immutable disjoint union. immutable=None (default) means that the graphs and their disjoint union will behave the same way.

See also:

- *union()*
- *join()*

EXAMPLES:

```
sage: G = graphs.CycleGraph(3)
sage: H = graphs.CycleGraph(4)
sage: J = G.disjoint_union(H); J
Cycle graph disjoint_union Cycle graph: Graph on 7 vertices
sage: J.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (1, 3)]
sage: J = G.disjoint_union(H, labels='integers'); J
Cycle graph disjoint_union Cycle graph: Graph on 7 vertices
sage: J.vertices()
[0, 1, 2, 3, 4, 5, 6]
sage: (G + H).vertices()  # '+'-operator is a shortcut
[0, 1, 2, 3, 4, 5, 6]
```

```
sage: G = Graph({'a': ['b']})
sage: G.name("Custom path")
sage: G.name()
'Custom path'
sage: H = graphs.CycleGraph(3)
sage: J = G.disjoint_union(H); J
Custom path disjoint_union Cycle graph: Graph on 5 vertices
sage: J.vertices()
[(0, 'a'), (0, 'b'), (1, 0), (1, 1), (1, 2)]
```

disjunctive_product(*other*)

Return the disjunctive product of `self` and `other`.

The disjunctive product of G and H is the graph L with vertex set $V(L) = V(G) \times V(H)$, and $((u, v), (w, x))$ is an edge iff either :

- (u, w) is an edge of G, or
- (v, x) is an edge of H.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: D = Z.disjunctive_product(Z); D
Graph on 4 vertices
sage: D.plot() # long time
Graphics object consisting of 11 graphics primitives
```

```
sage: C = graphs.CycleGraph(5)
sage: D = C.disjunctive_product(Z); D
Graph on 10 vertices
sage: D.plot() # long time
Graphics object consisting of 46 graphics primitives
```

distance(*u*, *v*, *by_weight=False*)

Return the (directed) distance from `u` to `v` in the (di)graph.

The distance is the length of the shortest path from `u` to `v`.

This method simply calls *shortest_path_length()*, with default arguments. For more information, and for more option, we refer to that method.

INPUT:

- `by_weight` – boolean (default: `False`); if `False`, the graph is considered unweighted, and the distance is the number of edges in a shortest path. If `True`, the distance is the sum of edge labels (which are assumed to be numbers).

EXAMPLES:

```
sage: G = graphs.CycleGraph(9)
sage: G.distance(0,1)
1
sage: G.distance(0,4)
4
sage: G.distance(0,5)
4
sage: G = Graph({0:[], 1:[]})
sage: G.distance(0,1)
+Infinity
```

(continues on next page)

(continued from previous page)

```
sage: G = Graph({ 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2}},␣
↪sparse = True)
sage: G.plot(edge_labels=True).show() # long time
sage: G.distance(0, 3)
2
sage: G.distance(0, 3, by_weight=True)
3
```

distance_all_pairs(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the distances between all pairs of vertices.

INPUT:

- `by_weight` boolean (default: $False$); if `True`, the edges in the graph are weighted; if `False`, all edges have weight 1.
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'BFS'`: the computation is done through a BFS centered on each vertex successively. Works only if `by_weight==False`.
  - `'Floyd-Warshall-Cython'`: the Cython implementation of the Floyd-Warshall algorithm. Works only if `by_weight==False`.
  - `'Floyd-Warshall-Python'`: the Python implementation of the Floyd-Warshall algorithm. Works also with weighted graphs, even with negative weights (but no negative cycle is allowed).
  - `'Dijkstra_NetworkX'`: the Dijkstra algorithm, implemented in NetworkX. It works with weighted graphs, but no negative weight is allowed.
  - `'Dijkstra_Boost'`: the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - `'Johnson_Boost'`: the Johnson algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - `None` (default): Sage chooses the best algorithm: `'BFS'` if `by_weight` is `False`, `'Dijkstra_Boost'` if all weights are positive, `'Floyd-Warshall-Cython'` otherwise.
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); whether to check that the `weight_function` outputs a number for each edge.

OUTPUT:

A doubly indexed dictionary

Note: There is a Cython version of this method that is usually much faster for large graphs, as most of the time is actually spent building the final double dictionary. Everything on the subject is to be found in the *distances_all_pairs* module.

Note: This algorithm simply calls *GenericGraph.shortest_path_all_pairs()*, and we suggest to look at that method for more information and examples.

EXAMPLES:

The Petersen Graph:

```
sage: g = graphs.PetersenGraph()
sage: print(g.distance_all_pairs())
{0: {0: 0, 1: 1, 2: 2, 3: 2, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2, 9: 2}, 1: {0: 1,␣
↪1: 0, 2: 1, 3: 2, 4: 2, 5: 2, 6: 1, 7: 2, 8: 2, 9: 2}, 2: {0: 2, 1: 1, 2: 0,
↪ 3: 1, 4: 2, 5: 2, 6: 2, 7: 1, 8: 2, 9: 2}, 3: {0: 2, 1: 2, 2: 1, 3: 0, 4:␣
↪1, 5: 2, 6: 2, 7: 2, 8: 1, 9: 2}, 4: {0: 1, 1: 2, 2: 2, 3: 1, 4: 0, 5: 2,␣
↪6: 2, 7: 2, 8: 2, 9: 1}, 5: {0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 0, 6: 2, 7: 1,
↪ 8: 1, 9: 2}, 6: {0: 2, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2, 6: 0, 7: 2, 8: 1, 9:␣
↪1}, 7: {0: 2, 1: 2, 2: 1, 3: 2, 4: 2, 5: 1, 6: 2, 7: 0, 8: 2, 9: 1}, 8: {0:␣
↪2, 1: 2, 2: 2, 3: 1, 4: 2, 5: 1, 6: 1, 7: 2, 8: 0, 9: 2}, 9: {0: 2, 1: 2,␣
↪2: 2, 3: 2, 4: 1, 5: 2, 6: 1, 7: 1, 8: 2, 9: 0}}
```

Testing on Random Graphs:

```
sage: g = graphs.RandomGNP(20,.3)
sage: distances = g.distance_all_pairs()
sage: all(g.distance(0,v) == distances[0][v] for v in g)
True
```

See also:

- *distance_matrix()*
- *shortest_path_all_pairs()*

distance_graph(*dist*)

Return the graph on the same vertex set as the original graph but vertices are adjacent in the returned graph if and only if they are at specified distances in the original graph.

INPUT:

- `dist` – a nonnegative integer or a list of nonnegative integers; specified distance(s) for the connecting vertices. `Infinity` may be used here to describe vertex pairs in separate components.

OUTPUT:

The returned value is an undirected graph. The vertex set is identical to the calling graph, but edges of the returned graph join vertices whose distance in the calling graph are present in the input `dist`. Loops will only be present if distance 0 is included. If the original graph has a position dictionary specifying locations of vertices for plotting, then this information is copied over to the distance graph. In some instances this layout may not be the best, and might even be confusing when edges run on top of each other due to symmetries chosen for the layout.

EXAMPLES:

```
sage: G = graphs.CompleteGraph(3)
sage: H = G.cartesian_product(graphs.CompleteGraph(2))
sage: K = H.distance_graph(2)
sage: K.am()
[0 0 0 1 0 1]
[0 0 1 0 1 0]
[0 1 0 0 0 1]
[1 0 0 0 1 0]
[0 1 0 1 0 0]
[1 0 1 0 0 0]
```

To obtain the graph where vertices are adjacent if their distance apart is `d` or less use a `range()` command to create the input, using `d + 1` as the input to `range`. Notice that this will include distance 0 and hence place a loop at each vertex. To avoid this, use `range(1, d + 1)`:

```
sage: G = graphs.OddGraph(4)
sage: d = G.diameter()
sage: n = G.num_verts()
sage: H = G.distance_graph(list(range(d+1)))
sage: H.is_isomorphic(graphs.CompleteGraph(n))
False
sage: H = G.distance_graph(list(range(1,d+1)))
sage: H.is_isomorphic(graphs.CompleteGraph(n))
True
```

A complete collection of distance graphs will have adjacency matrices that sum to the matrix of all ones:

```
sage: P = graphs.PathGraph(20)
sage: all_ones = sum([P.distance_graph(i).am() for i in range(20)])
sage: all_ones == matrix(ZZ, 20, 20, [1]*400)
True
```

Four-bit strings differing in one bit is the same as four-bit strings differing in three bits:

```
sage: G = graphs.CubeGraph(4)
sage: H = G.distance_graph(3)
sage: G.is_isomorphic(H)
True
```

The graph of eight-bit strings, adjacent if different in an odd number of bits:

```
sage: G = graphs.CubeGraph(8) # long time
sage: H = G.distance_graph([1,3,5,7]) # long time
sage: degrees = [0]*sum([binomial(8,j) for j in [1,3,5,7]]) # long time
sage: degrees.append(2^8) # long time
sage: degrees == H.degree_histogram() # long time
True
```

An example of using `Infinity` as the distance in a graph that is not connected:

```
sage: G = graphs.CompleteGraph(3)
sage: H = G.disjoint_union(graphs.CompleteGraph(2))
sage: L = H.distance_graph(Infinity)
sage: L.am()
[0 0 0 1 1]
[0 0 0 1 1]
[0 0 0 1 1]
[1 1 1 0 0]
[1 1 1 0 0]
```

AUTHOR:

Rob Beezer, 2009-11-25

distance_matrix(*vertices=None*, ***kwds*)

Return the distance matrix of (di)graph.

The (di)graph is expected to be (strongly) connected.

The distance matrix of a (strongly) connected (di)graph is a matrix whose rows and columns are by default (`vertices == None`) indexed with the positions of the vertices of the (di)graph in the ordering

vertices(). When vertices is set, the position of the vertices in this ordering is used. The intersection of row i and column j contains the shortest path distance from the vertex at the i-th position to the vertex at the j-th position.

Note that even when the vertices are consecutive integers starting from one, usually the vertex is not equal to its index.

INPUT:

- vertices – list (default: None); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *vertices()* is used. Because *vertices()* only works if the vertices can be sorted, using vertices is useful when working with possibly non-sortable objects in Python 3.
- All other arguments are forwarded to the subfunction *distance_all_pairs()*

EXAMPLES:

```
sage: d = DiGraph({1: [2, 3], 2: [3], 3: [4], 4: [1]})
sage: d.distance_matrix()
[0 1 1 2]
[3 0 1 2]
[2 3 0 1]
[1 2 2 0]
sage: d.distance_matrix(vertices=[4, 3, 2, 1])
[0 2 2 1]
[1 0 3 2]
[2 1 0 3]
[2 1 1 0]

sage: G = graphs.CubeGraph(3)
sage: G.distance_matrix()
[0 1 1 2 1 2 2 3]
[1 0 2 1 2 1 3 2]
[1 2 0 1 2 3 1 2]
[2 1 1 0 3 2 2 1]
[1 2 2 3 0 1 1 2]
[2 1 3 2 1 0 2 1]
[2 3 1 2 1 2 0 1]
[3 2 2 1 2 1 1 0]
```

The well known result of Graham and Pollak states that the determinant of the distance matrix of any tree of order n is $(-1)^{n-1}(n-1)2^{n-2}$:

```
sage: all(T.distance_matrix().det() == (-1)^9*(9)*2^8 for T in graphs.
↪trees(10))
True
```

See also:

- *distance_all_pairs()* – computes the distance between any two vertices.

distances_distribution(G)

Return the distances distribution of the (di)graph in a dictionary.

This method *ignores all edge labels*, so that the distance considered is the topological distance.

OUTPUT:

A dictionary `d` such that the number of pairs of vertices at distance `k` (if any) is equal to $d[k] \cdot |V(G)| \cdot (|V(G)| - 1)$.

Note: We consider that two vertices that do not belong to the same connected component are at infinite distance, and we do not take the trivial pairs of vertices (v, v) at distance 0 into account. Empty (di)graphs and (di)graphs of order 1 have no paths and so we return the empty dictionary `{}`.

EXAMPLES:

An empty Graph:

```
sage: g = Graph()
sage: g.distances_distribution()
{}
```

A Graph of order 1:

```
sage: g = Graph()
sage: g.add_vertex(1)
sage: g.distances_distribution()
{}
```

A Graph of order 2 without edge:

```
sage: g = Graph()
sage: g.add_vertices([1,2])
sage: g.distances_distribution()
{+Infinity: 1}
```

The Petersen Graph:

```
sage: g = graphs.PetersenGraph()
sage: g.distances_distribution()
{1: 1/3, 2: 2/3}
```

A graph with multiple disconnected components:

```
sage: g = graphs.PetersenGraph()
sage: g.add_edge('good','wine')
sage: g.distances_distribution()
{1: 8/33, 2: 5/11, +Infinity: 10/33}
```

The de Bruijn digraph dB(2,3):

```
sage: D = digraphs.DeBruijn(2,3)
sage: D.distances_distribution()
{1: 1/4, 2: 11/28, 3: 5/14}
```

dominating_set(*g*, *independent=False*, *total=False*, *value_only=False*, *solver=None*, *verbose=0*)

Return a minimum dominating set of the graph.

A minimum dominating set S of a graph G is a set of its vertices of minimal cardinality such that any vertex of G is in S or has one of its neighbors in S. See the Wikipedia article Dominating_set.

As an optimization problem, it can be expressed as:

$$\text{Minimize : } \sum_{v \in G} b_v$$

$$\text{Such that : } \forall v \in G, b_v + \sum_{(u,v) \in G.edges()} b_u \geq 1$$

$$\forall x \in G, b_x \text{ is a binary variable}$$

INPUT:

- `independent` – boolean (default: `False`); when `True`, computes a minimum independent dominating set, that is a minimum dominating set that is also an independent set (see also `independent_set()`)
- `total` – boolean (default: `False`); when `True`, computes a total dominating set (see the See the Wikipedia article Dominating_set)
- `value_only` – boolean (default: `False`); whether to only return the cardinality of the computed dominating set, or to return its list of vertices (default)
- `solver` – (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic illustration on a `PappusGraph`:

```
sage: g = graphs.PappusGraph()
sage: g.dominating_set(value_only=True)
5
```

If we build a graph from two disjoint stars, then link their centers we will find a difference between the cardinality of an independent set and a stable independent set:

```
sage: g = 2 * graphs.StarGraph(5)
sage: g.add_edge(0, 6)
sage: len(g.dominating_set())
2
sage: len(g.dominating_set(independent=True))
6
```

The total dominating set of the Petersen graph has cardinality 4:

```
sage: G = graphs.PetersenGraph()
sage: G.dominating_set(total=True, value_only=True)
4
```

The dominating set is calculated for both the directed and undirected graphs (modification introduced in trac ticket #17905):

```
sage: g = digraphs.Path(3)
sage: g.dominating_set(value_only=True)
2
sage: g = graphs.PathGraph(3)
sage: g.dominating_set(value_only=True)
1
```

dominator_tree(*g*, *root*, *return_dict=False*, *reverse=False*)

Use Boost to compute the dominator tree of `g`, rooted at `root`.

A node d dominates a node n if every path from the entry node `root` to n must go through d. The immediate dominator of a node n is the unique node that strictly dominates n but does not dominate any other node that dominates n. A dominator tree is a tree where each node's children are those nodes it immediately dominates. For more information, see the Wikipedia article Dominator_(graph_theory).

If the graph is connected and undirected, the parent of a vertex v is:

- the root if v is in the same biconnected component as the root;
- the first cut vertex in a path from v to the root, otherwise.

If the graph is not connected, the dominator tree of the whole graph is equal to the dominator tree of the connected component of the root.

If the graph is directed, computing a dominator tree is more complicated, and it needs time $O(m \log m)$, where m is the number of edges. The implementation provided by Boost is the most general one, so it needs time $O(m \log m)$ even for undirected graphs.

INPUT:

- `g` – the input Sage (Di)Graph
- `root` – the root of the dominator tree
- `return_dict` – boolean (default: `False`); if `True`, the function returns a dictionary associating to each vertex its parent in the dominator tree. If `False` (default), it returns the whole tree, as a `Graph` or a `DiGraph`.
- `reverse` – boolean (default: `False`); when set to `True`, computes the dominator tree in the reverse graph

OUTPUT:

The dominator tree, as a graph or as a dictionary, depending on the value of `return_dict`. If the output is a dictionary, it will contain `None` in correspondence of `root` and of vertices that are not reachable from `root`. If the output is a graph, it will not contain vertices that are not reachable from `root`.

EXAMPLES:

An undirected grid is biconnected, and its dominator tree is a star (everyone's parent is the root):

```
sage: g = graphs.GridGraph([2,2]).dominator_tree((0,0))
sage: g.to_dictionary()
{(0, 0): [(0, 1), (1, 0), (1, 1)], (0, 1): [(0, 0)], (1, 0): [(0, 0)], (1,␣
↪1): [(0, 0)]}
```

If the graph is made by two 3-cycles C_1, C_2 connected by an edge (v, w), with $v \in C_1$, $w \in C_2$, the cut vertices are v and w, the biconnected components are C_1, C_2, and the edge (v, w). If the root is in C_1, the parent of each vertex in C_1 is the root, the parent of w is v, and the parent of each vertex in C_2 is w:

```
sage: G = 2 * graphs.CycleGraph(3)
sage: v = 0
sage: w = 3
sage: G.add_edge(v,w)
sage: G.dominator_tree(1, return_dict=True)
{0: 1, 1: None, 2: 1, 3: 0, 4: 3, 5: 3}
```

An example with a directed graph:

```
sage: g = digraphs.Circuit(10).dominator_tree(5)
sage: g.to_dictionary()
{0: [1], 1: [2], 2: [3], 3: [4], 4: [], 5: [6], 6: [7], 7: [8], 8: [9], 9:␣
↪[0]}
sage: g = digraphs.Circuit(10).dominator_tree(5, reverse=True)
sage: g.to_dictionary()
{0: [9], 1: [0], 2: [1], 3: [2], 4: [3], 5: [4], 6: [], 7: [6], 8: [7], 9:␣
↪[8]}
```

If the output is a dictionary:

```
sage: graphs.GridGraph([2,2]).dominator_tree((0,0), return_dict=True)
{(0, 0): None, (0, 1): (0, 0), (1, 0): (0, 0), (1, 1): (0, 0)}
```

eccentricity(*v=None*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*, *dist_dict=None*, *with_labels=False*)

Return the eccentricity of vertex (or vertices) `v`.

The eccentricity of a vertex is the maximum distance to any other vertex.

For more information and examples on how to use input variables, see *shortest_paths()*

INPUT:

- `v` - either a single vertex or a list of vertices. If it is not specified, then it is taken to be all vertices.
- `by_weight` – boolean (default: `False`); if `True`, edge weights are taken into account; if False, all edges have weight 1
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'BFS'` - the computation is done through a BFS centered on each vertex successively. Works only if `by_weight==False`.
  - `'Floyd-Warshall-Cython'` - a Cython implementation of the Floyd-Warshall algorithm. Works only if `by_weight==False` and `v is None`.
  - `'Floyd-Warshall-Python'` - a Python implementation of the Floyd-Warshall algorithm. Works also with weighted graphs, even with negative weights (but no negative cycle is allowed). However, `v` must be `None`.
  - `'Dijkstra_NetworkX'` - the Dijkstra algorithm, implemented in NetworkX. It works with weighted graphs, but no negative weight is allowed.
  - `'Dijkstra_Boost'` - the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - `'Johnson_Boost'` - the Johnson algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - `'From_Dictionary'` - uses the (already computed) distances, that are provided by input variable `dist_dict`.
  - `None` (default): Sage chooses the best algorithm: `'From_Dictionary'` if `dist_dict` is not None, `'BFS'` for unweighted graphs, `'Dijkstra_Boost'` if all weights are positive, `'Johnson_Boost'` otherwise.
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); if `True`, we check that the `weight_function` outputs a number for each edge

- dist_dict – a dictionary (default: None); a dict of dicts of distances (used only if algorithm=='From_Dictionary')
- with_labels – boolean (default: False); whether to return a list or a dictionary keyed by vertices.

EXAMPLES:

```
sage: G = graphs.KrackhardtKiteGraph()
sage: G.eccentricity()
[4, 4, 4, 4, 4, 3, 3, 2, 3, 4]
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: G.eccentricity(7)
2
sage: G.eccentricity([7,8,9])
[2, 3, 4]
sage: G.eccentricity([7,8,9], with_labels=True) == {8: 3, 9: 4, 7: 2}
True
sage: G = Graph( { 0 : [], 1 : [], 2 : [1] } )
sage: G.eccentricity()
[+Infinity, +Infinity, +Infinity]
sage: G = Graph({0:[]})
sage: G.eccentricity(with_labels=True)
{0: 0}
sage: G = Graph({0:[], 1:[]})
sage: G.eccentricity(with_labels=True)
{0: +Infinity, 1: +Infinity}
sage: G = Graph([(0,1,1), (1,2,1), (0,2,3)])
sage: G.eccentricity(algorithm = 'BFS')
[1, 1, 1]
sage: G.eccentricity(algorithm = 'Floyd-Warshall-Cython')
[1, 1, 1]
sage: G.eccentricity(by_weight = True, algorithm = 'Dijkstra_NetworkX')
[2, 1, 2]
sage: G.eccentricity(by_weight = True, algorithm = 'Dijkstra_Boost')
[2, 1, 2]
sage: G.eccentricity(by_weight = True, algorithm = 'Johnson_Boost')
[2, 1, 2]
sage: G.eccentricity(by_weight = True, algorithm = 'Floyd-Warshall-Python')
[2, 1, 2]
sage: G.eccentricity(dist_dict = G.shortest_path_all_pairs(by_weight =␣
↪True)[0])
[2, 1, 2]
```

edge_boundary(*vertices1*, *vertices2=None*, *labels=True*, *sort=False*)

Return a list of edges (u,v,l) with u in vertices1 and v in vertices2.

If vertices2 is None, then it is set to the complement of vertices1.

In a digraph, the external boundary of a vertex v are those vertices u with an arc (v, u).

INPUT:

- labels – boolean (default: True); if False, each edge is a tuple (u, v) of vertices
- sort – boolean (default False); whether to sort the result

EXAMPLES:

```
sage: K = graphs.CompleteBipartiteGraph(9, 3)
sage: len(K.edge_boundary([0, 1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11]))
```

(continues on next page)

(continued from previous page)

```
27
sage: K.size()
27
```

Note that the edge boundary preserves direction:

```
sage: K = graphs.CompleteBipartiteGraph(9, 3).to_directed()
sage: len(K.edge_boundary([0, 1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11]))
27
sage: K.size()
54
```

```
sage: D = DiGraph({0: [1, 2], 3: [0]})
sage: D.edge_boundary([0], sort=True)
[(0, 1, None), (0, 2, None)]
sage: D.edge_boundary([0], labels=False, sort=True)
[(0, 1), (0, 2)]
```

edge_connectivity(*G*, *value_only=True*, *implementation=None*, *use_edge_labels=False*, *vertices=False*, *solver=None*, *verbose=0*)

Return the edge connectivity of the graph.

For more information, see the Wikipedia article Connectivity_(graph_theory).

Note: When the graph is a directed graph, this method actually computes the *strong* connectivity, (i.e. a directed graph is strongly k-connected if there are k disjoint paths between any two vertices u, v). If you do not want to consider strong connectivity, the best is probably to convert your `DiGraph` object to a `Graph` object, and compute the connectivity of this other graph.

INPUT:

- `G` – the input Sage (Di)Graph
- `value_only` – boolean (default: `True`)
  - When set to `True` (default), only the value is returned.
  - When set to `False`, both the value and a minimum vertex cut are returned.
- `implementation` – string (default: `None`); selects an implementation:
  - `None` (default) – selects the best implementation available
  - `"boost"` – use the Boost graph library (which is much more efficient). It is not available when `edge_labels=True`, and it is unreliable for directed graphs (see trac ticket #18753).
  - **`"Sage"` – use Sage's implementation based on integer linear** programming
- `use_edge_labels` – boolean (default: `False`)
  - When set to `True`, computes a weighted minimum cut where each edge has a weight defined by its label. (If an edge has no label, 1 is assumed.). Implies `boost` = `False`.
  - When set to `False`, each edge has weight 1.
- `vertices` – boolean (default: `False`)
  - When set to `True`, also returns the two sets of vertices that are disconnected by the cut. Implies `value_only=False`.

- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used (ignored if `implementation='boost'`). If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic application on the PappusGraph:

```
sage: from sage.graphs.connectivity import edge_connectivity
sage: g = graphs.PappusGraph()
sage: edge_connectivity(g)
3
sage: g.edge_connectivity()
3
```

The edge connectivity of a complete graph is its minimum degree, and one of the two parts of the bipartition is reduced to only one vertex. The graph of the cut edges is isomorphic to a Star graph:

```
sage: g = graphs.CompleteGraph(5)
sage: [ value, edges, [ setA, setB ]] = edge_connectivity(g,vertices=True)
sage: value
4
sage: len(setA) == 1 or len(setB) == 1
True
sage: cut = Graph()
sage: cut.add_edges(edges)
sage: cut.is_isomorphic(graphs.StarGraph(4))
True
```

Even if obviously in any graph we know that the edge connectivity is less than the minimum degree of the graph:

```
sage: g = graphs.RandomGNP(10,.3)
sage: min(g.degree()) >= edge_connectivity(g)
True
```

If we build a tree then assign to its edges a random value, the minimum cut will be the edge with minimum value:

```
sage: tree = graphs.RandomTree(10)
sage: for u,v in tree.edge_iterator(labels=None):
....:     tree.set_edge_label(u, v, random())
sage: minimum = min(tree.edge_labels())
sage: [_, [(_, _, l)]] = edge_connectivity(tree, value_only=False, use_edge_
↪labels=True)
sage: l == minimum
True
```

When `value_only=True` and `implementation="sage"`, this function is optimized for small connectivity values and does not need to build a linear program.

It is the case for graphs which are not connected

```
sage: g = 2 * graphs.PetersenGraph()
sage: edge_connectivity(g, implementation="sage")
0.0
```

For directed graphs, the strong connectivity is tested through the dedicated function:

```
sage: g = digraphs.ButterflyGraph(3)
sage: edge_connectivity(g, implementation="sage")
0.0
```

We check that the result with Boost is the same as the result without Boost:

```
sage: g = graphs.RandomGNP(15, .3)
sage: edge_connectivity(g, implementation="boost") == edge_connectivity(g,␣
↪implementation="sage")
True
```

Boost interface also works with directed graphs:

```
sage: edge_connectivity(digraphs.Circuit(10), implementation="boost",␣
↪vertices=True)
[1, [(0, 1)], [{0}, {1, 2, 3, 4, 5, 6, 7, 8, 9}]]
```

However, the Boost algorithm is not reliable if the input is directed (see trac ticket #18753):

```
sage: g = digraphs.Path(3)
sage: edge_connectivity(g)
0.0
sage: edge_connectivity(g, implementation="boost")
1
sage: g.add_edge(1, 0)
sage: edge_connectivity(g)
0.0
sage: edge_connectivity(g, implementation="boost")
0
```

edge_cut(*s*, *t*, *value_only=True*, *use_edge_labels=False*, *vertices=False*, *algorithm='FF'*, *solver=None*, *verbose=0*)

Return a minimum edge cut between vertices s and t.

A minimum edge cut between two vertices s and t of self is a set A of edges of minimum weight such that the graph obtained by removing A from the graph is disconnected. For more information, see the Wikipedia article Cut_(graph_theory).

INPUT:

- `s` – source vertex
- `t` – sink vertex
- `value_only` – boolean (default: `True`); whether to return only the weight of a minimum cut (`True`) or a list of edges of a minimum cut (`False`)
- `use_edge_labels` – boolean (default: `False`); whether to compute a weighted minimum edge cut where the weight of an edge is defined by its label (if an edge has no label, 1 is assumed), or to compute a cut of minimum cardinality (i.e., edge weights are set to 1)
- `vertices` – boolean (default: `False`); whether set to `True`, return a list of edges in the edge cut and the two sets of vertices that are disconnected by the cut

  Note: `vertices=True` implies `value_only=False`.
- `algorithm` – string (default: `'FF'`); algorithm to use:
  - If `algorithm = "FF"`, a Python implementation of the Ford-Fulkerson algorithm is used

  - If `algorithm = "LP"`, the problem is solved using Linear Programming.
  - If `algorithm = "igraph"`, the igraph implementation of the Goldberg-Tarjan algorithm is used (only available when `igraph` is installed)
  - If `algorithm = None`, the problem is solved using the default maximum flow algorithm (see *flow()*)

- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: The use of Linear Programming for non-integer problems may possibly mean the presence of a (slight) numerical noise.

OUTPUT:

Real number or tuple, depending on the given arguments (examples are given below).

EXAMPLES:

A basic application in the Pappus graph:

```
sage: g = graphs.PappusGraph()
sage: g.edge_cut(1, 2, value_only=True)
3
```

Or on Petersen's graph, with the corresponding bipartition of the vertex set:

```
sage: g = graphs.PetersenGraph()
sage: g.edge_cut(0, 3, vertices=True)
[3, [(0, 1, None), (0, 4, None), (0, 5, None)], [[0], [1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]]
```

If the graph is a path with randomly weighted edges:

```
sage: g = graphs.PathGraph(15)
sage: for u,v in g.edge_iterator(labels=None):
....:     g.set_edge_label(u, v, random())
```

The edge cut between the two ends is the edge of minimum weight:

```
sage: minimum = min(g.edge_labels())
sage: minimum == g.edge_cut(0, 14, use_edge_labels=True)
True
sage: [value, [e]] = g.edge_cut(0, 14, use_edge_labels=True, value_only=False)
sage: g.edge_label(e[0], e[1]) == minimum
True
```

The two sides of the edge cut are obviously shorter paths:

```
sage: value,edges,[set1,set2] = g.edge_cut(0, 14, use_edge_labels=True,␣
↪vertices=True)
sage: g.subgraph(set1).is_isomorphic(graphs.PathGraph(len(set1)))
True
sage: g.subgraph(set2).is_isomorphic(graphs.PathGraph(len(set2)))
```

(continues on next page)

(continued from previous page)

```
True
sage: len(set1) + len(set2) == g.order()
True
```

edge_disjoint_paths(*s*, *t*, *algorithm='FF'*, *solver=None*, *verbose=False*)

Return a list of edge-disjoint paths between two vertices.

The edge version of Menger's theorem asserts that the size of the minimum edge cut between two vertices s and't' (the minimum number of edges whose removal disconnects s and t) is equal to the maximum number of pairwise edge-independent paths from s to t.

This function returns a list of such paths.

INPUT:

- `algorithm` – string (default: `"FF"`); the algorithm to use among:
  - `"FF"`, a Python implementation of the Ford-Fulkerson algorithm
  - `"LP"`, the flow problem is solved using Linear Programming
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: This function is topological: it does not take the eventual weights of the edges into account.

EXAMPLES:

In a complete bipartite graph

```
sage: g = graphs.CompleteBipartiteGraph(2, 3)
sage: g.edge_disjoint_paths(0, 1)
[[0, 2, 1], [0, 3, 1], [0, 4, 1]]
```

edge_disjoint_spanning_trees(*k*, *root=None*, *solver=None*, *verbose=0*)

Return the desired number of edge-disjoint spanning trees/arborescences.

INPUT:

- `k` – integer; the required number of edge-disjoint spanning trees/arborescences
- `root` – vertex (default: `None`); root of the disjoint arborescences when the graph is directed. If set to `None`, the first vertex in the graph is picked.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

ALGORITHM:

Mixed Integer Linear Program. The formulation can be found in [Coh2019].

There are at least two possible rewritings of this method which do not use Linear Programming:

- The algorithm presented in the paper entitled "A short proof of the tree-packing theorem", by Thomas Kaiser [Kai2012].

- The implementation of a Matroid class and of the Matroid Union Theorem (see section 42.3 of [Sch2003]), applied to the cycle Matroid (see chapter 51 of [Sch2003]).

EXAMPLES:

The Petersen Graph does have a spanning tree (it is connected):

```
sage: g = graphs.PetersenGraph()
sage: [T] = g.edge_disjoint_spanning_trees(1)
sage: T.is_tree()
True
```

Though, it does not have 2 edge-disjoint trees (as it has less than $2(|V|-1)$ edges):

```
sage: g.edge_disjoint_spanning_trees(2)
Traceback (most recent call last):
...
EmptySetError: this graph does not contain the required number of trees/
↪arborescences
```

By Edmond's theorem, a graph which is k-connected always has k edge-disjoint arborescences, regardless of the root we pick:

```
sage: g = digraphs.RandomDirectedGNP(28, .3) # reduced from 30 to 28, cf.
↪#9584
sage: k = Integer(g.edge_connectivity())
sage: arborescences = g.edge_disjoint_spanning_trees(k)  # long time (up to␣
↪15s on sage.math, 2011)
sage: all(a.is_directed_acyclic() for a in arborescences)  # long time
True
sage: all(a.is_connected() for a in arborescences)  # long time
True
```

In the undirected case, we can only ensure half of it:

```
sage: g = graphs.RandomGNP(30, .3)
sage: k = Integer(g.edge_connectivity()) // 2
sage: trees = g.edge_disjoint_spanning_trees(k)
sage: all(t.is_tree() for t in trees)
True
```

edge_iterator(*vertices=None*, *labels=True*, *ignore_direction=False*)

Return an iterator over edges.

The iterator returned is over the edges incident with any vertex given in the parameter `vertices`. If the graph is directed, iterates over edges going out only. If `vertices` is `None`, then returns an iterator over all edges. If `self` is directed, returns outgoing edges only.

INPUT:

- `vertices` – object (default: `None`); a vertex, a list of vertices or `None`
- **`labels` – boolean (defaul: `True`); if `False`, each edge is a** tuple (u, v) of vertices
- **`ignore_direction` – boolean (defaul: `False`); only applies to** directed graphs. If `True`, searches across edges in either direction.

EXAMPLES:

```
sage: for i in graphs.PetersenGraph().edge_iterator([0]):
....:  print(i)
(0, 1, None)
(0, 4, None)
(0, 5, None)
sage: D = DiGraph({0: [1, 2], 1: [0]})
sage: for i in D.edge_iterator([0]):
....:  print(i)
(0, 1, None)
(0, 2, None)
```

```
sage: G = graphs.TetrahedralGraph()
sage: list(G.edge_iterator(labels=False))
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

```
sage: D = DiGraph({1: [0], 2: [0]})
sage: list(D.edge_iterator(0))
[]
sage: list(D.edge_iterator(0, ignore_direction=True))
[(1, 0, None), (2, 0, None)]
```

edge_label(*u*, *v*)

Return the label of an edge.

If the graph allows multiple edges, then the list of labels on the edges is returned.

See also:

- *set_edge_label()*

EXAMPLES:

```
sage: G = Graph({0: {1: 'edgelabel'}})
sage: G.edge_label(0, 1)
'edgelabel'
sage: D = DiGraph({1: {2: 'up'}, 2: {1: 'down'}})
sage: D.edge_label(2, 1)
'down'
```

```
sage: G = Graph(multiedges=True)
sage: [G.add_edge(0, 1, i) for i in range(1, 6)]
[None, None, None, None, None]
sage: sorted(G.edge_label(0, 1))
[1, 2, 3, 4, 5]
```

edge_labels()

Return a list of the labels of all edges in `self`.

The output list is not sorted.

EXAMPLES:

```
sage: G = Graph({0: {1: 'x', 2: 'z', 3: 'a'}, 2: {5: 'out'}}, sparse=True)
sage: G.edge_labels()
['x', 'z', 'a', 'out']
sage: G = DiGraph({0: {1: 'x', 2: 'z', 3: 'a'}, 2: {5: 'out'}}, sparse=True)
```

(continues on next page)

(continued from previous page)

```
sage: G.edge_labels()
['x', 'z', 'a', 'out']
```

edges(*labels=True*, *sort=True*, *key=None*)

Return a *EdgesView* of edges.

Each edge is a triple `(u, v, l)` where `u` and `v` are vertices and `l` is a label. If the parameter `labels` is `False` then a list of couple `(u, v)` is returned where `u` and `v` are vertices.

INPUT:

- `labels` – boolean (default: `True`); if `False`, each edge is simply a pair `(u, v)` of vertices
- `sort` – boolean (default: `True`); if `True`, edges are sorted according to the default ordering
- `key` – a function (default: `None`); a function that takes an edge (a pair or a triple, according to the `labels` keyword) as its one argument and returns a value that can be used for comparisons in the sorting algorithm

OUTPUT: A *EdgesView*.

Warning: Since any object may be a vertex, there is no guarantee that any two vertices will be comparable, and thus no guarantee how two edges may compare. With default objects for vertices (all integers), or when all the vertices are of the same simple type, then there should not be a problem with how the vertices will be sorted. However, if you need to guarantee a total order for the sorting of the edges, use the `key` argument, as illustrated in the examples below.

EXAMPLES:

```
sage: graphs.DodecahedralGraph().edges()
[(0, 1, None), (0, 10, None), (0, 19, None), (1, 2, None), (1, 8, None), (2,␣
↪3, None), (2, 6, None), (3, 4, None), (3, 19, None), (4, 5, None), (4, 17,␣
↪None), (5, 6, None), (5, 15, None), (6, 7, None), (7, 8, None), (7, 14,␣
↪None), (8, 9, None), (9, 10, None), (9, 13, None), (10, 11, None), (11, 12,␣
↪None), (11, 18, None), (12, 13, None), (12, 16, None), (13, 14, None), (14,␣
↪15, None), (15, 16, None), (16, 17, None), (17, 18, None), (18, 19, None)]
```

```
sage: graphs.DodecahedralGraph().edges(labels=False)
[(0, 1), (0, 10), (0, 19), (1, 2), (1, 8), (2, 3), (2, 6), (3, 4), (3, 19),␣
↪(4, 5), (4, 17), (5, 6), (5, 15), (6, 7), (7, 8), (7, 14), (8, 9), (9, 10),␣
↪(9, 13), (10, 11), (11, 12), (11, 18), (12, 13), (12, 16), (13, 14), (14,␣
↪15), (15, 16), (16, 17), (17, 18), (18, 19)]
```

```
sage: D = graphs.DodecahedralGraph().to_directed()
sage: D.edges()
[(0, 1, None), (0, 10, None), (0, 19, None), (1, 0, None), (1, 2, None), (1,␣
↪8, None), (2, 1, None), (2, 3, None), (2, 6, None), (3, 2, None), (3, 4,␣
↪None), (3, 19, None), (4, 3, None), (4, 5, None), (4, 17, None), (5, 4,␣
↪None), (5, 6, None), (5, 15, None), (6, 2, None), (6, 5, None), (6, 7,␣
↪None), (7, 6, None), (7, 8, None), (7, 14, None), (8, 1, None), (8, 7,␣
↪None), (8, 9, None), (9, 8, None), (9, 10, None), (9, 13, None), (10, 0,␣
↪None), (10, 9, None), (10, 11, None), (11, 10, None), (11, 12, None), (11,␣
↪18, None), (12, 11, None), (12, 13, None), (12, 16, None), (13, 9, None),␣
↪(13, 12, None), (13, 14, None), (14, 7, None), (14, 13, None), (14, 15,␣
↪None), (15, 5, None), (15, 14, None), (15, 16, None), (16, 12, None), (16,␣
↪15, None), (16, 17, None), (17, 4, None), (17, 16, None), (17, 18, None),␣
↪(18, 11, None), (18, 17, None), (18, 19, None), (19, 0, None), (19, 5,␣
↪None), (19, 18, None)]
```

(continues on next page)

(continued from previous page)

```
sage: D.edges(labels=False)
[(0, 1), (0, 10), (0, 19), (1, 0), (1, 2), (1, 8), (2, 1), (2, 3), (2, 6), (3,
↪ 2), (3, 4), (3, 19), (4, 3), (4, 5), (4, 17), (5, 4), (5, 6), (5, 15), (6,␣
↪2), (6, 5), (6, 7), (7, 6), (7, 8), (7, 14), (8, 1), (8, 7), (8, 9), (9, 8),
↪ (9, 10), (9, 13), (10, 0), (10, 9), (10, 11), (11, 10), (11, 12), (11, 18),
↪ (12, 11), (12, 13), (12, 16), (13, 9), (13, 12), (13, 14), (14, 7), (14,␣
↪13), (14, 15), (15, 5), (15, 14), (15, 16), (16, 12), (16, 15), (16, 17),␣
↪(17, 4), (17, 16), (17, 18), (18, 11), (18, 17), (18, 19), (19, 0), (19, 3),
↪ (19, 18)]
```

The default is to sort the returned list in the default fashion, as in the above examples. This can be overridden by specifying a key function. This first example just ignores the labels in the third component of the triple:

```
sage: G = graphs.CycleGraph(5)
sage: G.edges(key=lambda x: (x[1], -x[0]))
[(0, 1, None), (1, 2, None), (2, 3, None), (3, 4, None), (0, 4, None)]
```

We set the labels to characters and then perform a default sort followed by a sort according to the labels:

```
sage: G = graphs.CycleGraph(5)
sage: for e in G.edges(sort=False):
....:     G.set_edge_label(e[0], e[1], chr(ord('A') + e[0] + 5 * e[1]))
sage: G.edges(sort=True)
[(0, 1, 'F'), (0, 4, 'U'), (1, 2, 'L'), (2, 3, 'R'), (3, 4, 'X')]
sage: G.edges(key=lambda x: x[2])
[(0, 1, 'F'), (1, 2, 'L'), (2, 3, 'R'), (0, 4, 'U'), (3, 4, 'X')]
```

edges_incident (*vertices=None*, *labels=True*, *sort=False*)

Return incident edges to some vertices.

If `vertices` is a vertex, then it returns the list of edges incident to that vertex. If ``vertices`` is a list of vertices then it returns the list of all edges adjacent to those vertices. If `vertices` is `None`, it returns a list of all edges in graph. For digraphs, only lists outward edges.

INPUT:

- `vertices` – object (default: `None`); a vertex, a list of vertices or `None`
- **`labels` – boolean (default: `True`); if `False`, each edge is** a tuple (u, v) of vertices
- `sort` – boolean (default: `False`); if `True` the returned list is sorted

EXAMPLES:

```
sage: graphs.PetersenGraph().edges_incident([0, 9], labels=False)
[(0, 1), (0, 4), (0, 5), (4, 9), (6, 9), (7, 9)]
sage: D = DiGraph({0: [1]})
sage: D.edges_incident([0])
[(0, 1, None)]
sage: D.edges_incident([1])
[]
```

eigenspaces (*laplacian=False*)

Return the *right* eigenspaces of the adjacency matrix of the graph.

INPUT:

- **laplacian – boolean (default: False); if True, use the** Laplacian matrix (see *kirchhoff_matrix()*)

OUTPUT:

A list of pairs. Each pair is an eigenvalue of the adjacency matrix of the graph, followed by the vector space that is the eigenspace for that eigenvalue, when the eigenvectors are placed on the right of the matrix.

For some graphs, some of the eigenspaces are described exactly by vector spaces over a NumberField(). For numerical eigenvectors use *eigenvectors()*.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.eigenspaces()
[
(3, Vector space of degree 10 and dimension 1 over Rational Field
User basis matrix:
[1 1 1 1 1 1 1 1 1 1]),
(-2, Vector space of degree 10 and dimension 4 over Rational Field
User basis matrix:
[ 1  0  0  0 -1 -1 -1  0  1  1]
[ 0  1  0  0 -1  0 -2 -1  1  2]
[ 0  0  1  0 -1  1 -1 -2  0  2]
[ 0  0  0  1 -1  1  0 -1 -1  1]),
(1, Vector space of degree 10 and dimension 5 over Rational Field
User basis matrix:
[ 1  0  0  0  0  1 -1  0  0 -1]
[ 0  1  0  0  0 -1  1 -1  0  0]
[ 0  0  1  0  0  0 -1  1 -1  0]
[ 0  0  0  1  0  0  0 -1  1 -1]
[ 0  0  0  0  1 -1  0  0 -1  1])
]
```

Eigenspaces for the Laplacian should be identical since the Petersen graph is regular. However, since the output also contains the eigenvalues, the two outputs are slightly different:

```
sage: P.eigenspaces(laplacian=True)
[
(0, Vector space of degree 10 and dimension 1 over Rational Field
User basis matrix:
[1 1 1 1 1 1 1 1 1 1]),
(5, Vector space of degree 10 and dimension 4 over Rational Field
User basis matrix:
[ 1  0  0  0 -1 -1 -1  0  1  1]
[ 0  1  0  0 -1  0 -2 -1  1  2]
[ 0  0  1  0 -1  1 -1 -2  0  2]
[ 0  0  0  1 -1  1  0 -1 -1  1]),
(2, Vector space of degree 10 and dimension 5 over Rational Field
User basis matrix:
[ 1  0  0  0  0  1 -1  0  0 -1]
[ 0  1  0  0  0 -1  1 -1  0  0]
[ 0  0  1  0  0  0 -1  1 -1  0]
[ 0  0  0  1  0  0  0 -1  1 -1]
[ 0  0  0  0  1 -1  0  0 -1  1])
]
```

Notice how one eigenspace below is described with a square root of 2. For the two possible values (positive and negative) there is a corresponding eigenspace:

```
sage: C = graphs.CycleGraph(8)
sage: C.eigenspaces()
[
(2, Vector space of degree 8 and dimension 1 over Rational Field
User basis matrix:
[1 1 1 1 1 1 1 1]),
(-2, Vector space of degree 8 and dimension 1 over Rational Field
User basis matrix:
[ 1 -1  1 -1  1 -1  1 -1]),
(0, Vector space of degree 8 and dimension 2 over Rational Field
User basis matrix:
[ 1  0 -1  0  1  0 -1  0]
[ 0  1  0 -1  0  1  0 -1]),
(a3, Vector space of degree 8 and dimension 2 over Number Field in a3 with␣
↪defining polynomial x^2 - 2
User basis matrix:
[  1   0  -1 -a3  -1   0   1  a3]
[  0   1  a3   1   0  -1 -a3  -1])
]
```

A digraph may have complex eigenvalues and eigenvectors. For a 3-cycle, we have:

```
sage: T = DiGraph({0: [1], 1: [2], 2: [0]})
sage: T.eigenspaces()
[
(1, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[1 1 1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 + x + 1
User basis matrix:
[      1      a1 -a1 - 1])
]
```

eigenvectors(*laplacian=False*)

Return the *right* eigenvectors of the adjacency matrix of the graph.

INPUT:

- `laplacian` – boolean (default: `False`); if `True`, use the Laplacian matrix (see *kirchhoff_matrix()*)

OUTPUT:

A list of triples. Each triple begins with an eigenvalue of the adjacency matrix of the graph. This is followed by a list of eigenvectors for the eigenvalue, when the eigenvectors are placed on the right side of the matrix. Together, the eigenvectors form a basis for the eigenspace. The triple concludes with the algebraic multiplicity of the eigenvalue.

For some graphs, the exact eigenspaces provided by *eigenspaces()* provide additional insight into the structure of the eigenspaces.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.eigenvectors()
[(3, [
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
], 1), (-2, [
```

(continues on next page)

(continued from previous page)

```
(1, 0, 0, 0, -1, -1, -1, 0, 1, 1),
(0, 1, 0, 0, -1, 0, -2, -1, 1, 2),
(0, 0, 1, 0, -1, 1, -1, -2, 0, 2),
(0, 0, 0, 1, -1, 1, 0, -1, -1, 1)
], 4), (1, [
(1, 0, 0, 0, 0, 1, -1, 0, 0, -1),
(0, 1, 0, 0, 0, -1, 1, -1, 0, 0),
(0, 0, 1, 0, 0, 0, -1, 1, -1, 0),
(0, 0, 0, 1, 0, 0, 0, -1, 1, -1),
(0, 0, 0, 0, 1, -1, 0, 0, -1, 1)
], 5)]
```

Eigenspaces for the Laplacian should be identical since the Petersen graph is regular. However, since the output also contains the eigenvalues, the two outputs are slightly different:

```
sage: P.eigenvectors(laplacian=True)
[(0, [
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
], 1), (5, [
(1, 0, 0, 0, -1, -1, -1, 0, 1, 1),
(0, 1, 0, 0, -1, 0, -2, -1, 1, 2),
(0, 0, 1, 0, -1, 1, -1, -2, 0, 2),
(0, 0, 0, 1, -1, 1, 0, -1, -1, 1)
], 4), (2, [
(1, 0, 0, 0, 0, 1, -1, 0, 0, -1),
(0, 1, 0, 0, 0, -1, 1, -1, 0, 0),
(0, 0, 1, 0, 0, 0, -1, 1, -1, 0),
(0, 0, 0, 1, 0, 0, 0, -1, 1, -1),
(0, 0, 0, 0, 1, -1, 0, 0, -1, 1)
], 5)]
```

```
sage: C = graphs.CycleGraph(8)
sage: C.eigenvectors()
[(2, [
(1, 1, 1, 1, 1, 1, 1, 1)
], 1), (-2, [
(1, -1, 1, -1, 1, -1, 1, -1)
], 1), (0, [
(1, 0, -1, 0, 1, 0, -1, 0),
(0, 1, 0, -1, 0, 1, 0, -1)
], 2), (-1.4142135623..., [(1, 0, -1, 1.4142135623..., -1, 0, 1, -1.
↪4142135623...), (0, 1, -1.4142135623..., 1, 0, -1, 1.4142135623..., -1)],␣
↪2), (1.4142135623..., [(1, 0, -1, -1.4142135623..., -1, 0, 1, 1.4142135623..
↪.), (0, 1, 1.4142135623..., 1, 0, -1, -1.4142135623..., -1)], 2)]
```

A digraph may have complex eigenvalues. Previously, the complex parts of graph eigenvalues were being dropped. For a 3-cycle, we have:

```
sage: T = DiGraph({0:[1], 1:[2], 2:[0]})
sage: T.eigenvectors()
[(1, [
(1, 1, 1)
], 1), (-0.5000000000... - 0.8660254037...*I, [(1, -0.5000000000... - 0.
↪8660254037...*I, -0.5000000000... + 0.8660254037...*I)], 1), (-0.5000000000.
↪.. + 0.8660254037...*I, [(1, -0.5000000000... + 0.8660254037...*I, -0.
↪5000000000... - 0.8660254037...*I)], 1)]
```

eulerian_circuit(*return_vertices=False*, *labels=True*, *path=False*)

Return a list of edges forming an Eulerian circuit if one exists.

If no Eulerian circuit is found, the method returns `False`.

This is implemented using Hierholzer's algorithm.

INPUT:

- **return_vertices – boolean (default: False); optionally** provide a list of vertices for the path
- **labels – boolean (default: True); whether to return edges** with labels (3-tuples)
- **path – boolean (default: False); find an Eulerian path** instead

OUTPUT:

either ([edges], [vertices]) or [edges] of an Eulerian circuit (or path)

EXAMPLES:

```
sage: g = graphs.CycleGraph(5)
sage: g.eulerian_circuit()
[(0, 4, None), (4, 3, None), (3, 2, None), (2, 1, None), (1, 0, None)]
sage: g.eulerian_circuit(labels=False)
[(0, 4), (4, 3), (3, 2), (2, 1), (1, 0)]
```

```
sage: g = graphs.CompleteGraph(7)
sage: edges, vertices = g.eulerian_circuit(return_vertices=True)
sage: vertices
[0, 6, 5, 4, 6, 3, 5, 2, 4, 3, 2, 6, 1, 5, 0, 4, 1, 3, 0, 2, 1, 0]
```

```
sage: graphs.CompleteGraph(4).eulerian_circuit()
False
```

A disconnected graph can be Eulerian:

```
sage: g = Graph({0: [], 1: [2], 2: [3], 3: [1], 4: []})
sage: g.eulerian_circuit(labels=False)
[(1, 3), (3, 2), (2, 1)]
```

```
sage: g = DiGraph({0: [1], 1: [2, 4], 2:[3], 3:[1]})
sage: g.eulerian_circuit(labels=False, path=True)
[(0, 1), (1, 2), (2, 3), (3, 1), (1, 4)]
```

```
sage: g = Graph({0:[1,2,3], 1:[2,3], 2:[3,4], 3:[4]})
sage: g.is_eulerian(path=True)
(0, 1)
sage: g.eulerian_circuit(labels=False, path=True)
[(1, 3), (3, 4), (4, 2), (2, 3), (3, 0), (0, 2), (2, 1), (1, 0)]
```

eulerian_orientation()

Return a DiGraph which is an Eulerian orientation of the current graph.

An Eulerian graph being a graph such that any vertex has an even degree, an Eulerian orientation of a graph is an orientation of its edges such that each vertex v verifies $d^+(v) = d^-(v) = d(v)/2$, where d^+ and d^- respectively represent the out-degree and the in-degree of a vertex.

If the graph is not Eulerian, the orientation verifies for any vertex v that $|d^+(v) - d^-(v)| \leq 1$.

ALGORITHM:

This algorithm is a random walk through the edges of the graph, which orients the edges according to the walk. When a vertex is reached which has no non-oriented edge (this vertex must have odd degree), the walk resumes at another vertex of odd degree, if any.

This algorithm has complexity $O(m)$, where m is the number of edges in the graph.

EXAMPLES:

The CubeGraph with parameter 4, which is regular of even degree, has an Eulerian orientation such that $d^+ = d^-$:

```
sage: g = graphs.CubeGraph(4)
sage: g.degree()
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
sage: o = g.eulerian_orientation()
sage: o.in_degree()
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
sage: o.out_degree()
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Secondly, the Petersen Graph, which is 3 regular has an orientation such that the difference between d^+ and d^- is at most 1:

```
sage: g = graphs.PetersenGraph()
sage: o = g.eulerian_orientation()
sage: o.in_degree()
[2, 2, 2, 2, 2, 1, 1, 1, 1, 1]
sage: o.out_degree()
[1, 1, 1, 1, 1, 2, 2, 2, 2, 2]
```

export_to_file (*filename*, *format=None*, ***kwds*)

Export the graph to a file.

INPUT:

- `filename` – string; a file name
- `format` – string (default: `None`); select the output format explicitly. If set to `None` (default), the format is set to be the file extension of `filename`. Admissible formats are: `adjlist`, `dot`, `edgelist`, `gexf`, `gml`, `graphml`, `multiline_adjlist`, `pajek`, `yaml`.
- All other arguments are forwarded to the subfunction. For more information, see their respective documentation:

adjlist	http://networkx.lanl.gov/reference/generated/networkx.readwrite.adjlist.write_adjlist.html
dot	https://networkx.github.io/documentation/latest/reference/generated/networkx.drawing.nx_pydot.write_dot.html
edgelist	http://networkx.lanl.gov/reference/generated/networkx.readwrite.edgelist.write_edgelist.html
gexf	http://networkx.lanl.gov/reference/generated/networkx.readwrite.gexf.write_gexf.html
gml	http://networkx.lanl.gov/reference/generated/networkx.readwrite.gml.write_gml.html
graphml	http://networkx.lanl.gov/reference/generated/networkx.readwrite.graphml.write_graphml.html
multiline_adjlist	http://networkx.lanl.gov/reference/generated/networkx.readwrite.multiline_adjlist.write_multiline_adjlist.html
pajek	http://networkx.lanl.gov/reference/generated/networkx.readwrite.pajek.write_pajek.html
yaml	http://networkx.lanl.gov/reference/generated/networkx.readwrite.nx_yaml.write_yaml.html

See also:

- `save()` – save a Sage object to a 'sobj' file (preserves all its attributes)

Note: This functions uses the `write_*` functions defined in NetworkX (see http://networkx.lanl.gov/reference/readwrite.html).

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: filename = tmp_filename(ext=".pajek")
sage: g.export_to_file(filename)
sage: import networkx
sage: G_networkx = networkx.read_pajek(filename)
sage: Graph(G_networkx).is_isomorphic(g)
True
sage: filename = tmp_filename(ext=".edgelist")
sage: g.export_to_file(filename, data=False)
sage: h = Graph(networkx.read_edgelist(filename))
sage: g.is_isomorphic(h)
True
```

faces(*embedding=None*)

Return the faces of an embedded graph.

A combinatorial embedding of a graph is a clockwise ordering of the neighbors of each vertex. From this information one can define the faces of the embedding, which is what this method returns.

INPUT:

- `embedding` – dictionary (default: `None`); a combinatorial embedding dictionary. Format: `{v1: [v2,v3], v2: [v1], v3: [v1]}` (clockwise ordering of neighbors at each vertex). If set to `None` (default) the method will use the embedding stored as `self._embedding`. If none is stored, the method will compute the set of faces from the embedding returned by *`is_planar()`* (if the graph is, of course, planar).

Note: `embedding` is an ordered list based on the hash order of the vertices of graph. To avoid confusion, it might be best to set the rot_sys based on a 'nice_copy' of the graph.

See also:

- *set_embedding()*
- *get_embedding()*
- *is_planar()*
- *planar_dual()*

EXAMPLES:

Providing an embedding:

```
sage: T = graphs.TetrahedralGraph()
sage: T.faces({0: [1, 3, 2], 1: [0, 2, 3], 2: [0, 3, 1], 3: [0, 1, 2]})
[[(0, 1), (1, 2), (2, 0)],
 [(3, 2), (2, 1), (1, 3)],
 [(3, 0), (0, 2), (2, 3)],
 [(3, 1), (1, 0), (0, 3)]]
```

With no embedding provided:

```
sage: graphs.TetrahedralGraph().faces()
[[(0, 1), (1, 2), (2, 0)],
 [(3, 2), (2, 1), (1, 3)],
 [(3, 0), (0, 2), (2, 3)],
 [(3, 1), (1, 0), (0, 3)]]
```

With no embedding provided (non-planar graph):

```
sage: graphs.PetersenGraph().faces()
Traceback (most recent call last):
...
ValueError: no embedding is provided and the graph is not planar
```

feedback_vertex_set(*value_only=False*, *solver=None*, *verbose=0*, *constraint_generation=True*)

Return the minimum feedback vertex set of a (di)graph.

The minimum feedback vertex set of a (di)graph is a set of vertices that intersect all of its cycles. Equivalently, a minimum feedback vertex set of a (di)graph is a set S of vertices such that the digraph $G - S$ is acyclic. For more information, see the Wikipedia article Feedback_vertex_set.

INPUT:

- `value_only` – boolean (default: `False`); whether to return only the minimum cardinal of a minimum vertex set, or the `Set` of vertices of a minimal feedback vertex set
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.
- `constraint_generation` – boolean (default: `True`); whether to use constraint generation when solving the Mixed Integer Linear Program

ALGORITHMS:

(Constraints generation)

When the parameter `constraint_generation` is enabled (default) the following MILP formulation is used to solve the problem:

$$\text{Minimize : } \sum_{v \in G} b_v$$

$$\text{Such that :}$$

$$\forall C \text{ circuits } \subseteq G, \sum_{v \in C} b_v \geq 1$$

As the number of circuits contained in a graph is exponential, this LP is solved through constraint generation. This means that the solver is sequentially asked to solve the problem, knowing only a portion of the circuits contained in G, each time adding to the list of its constraints the circuit which its last answer had left intact.

(Another formulation based on an ordering of the vertices)

When the graph is directed, a second (and very slow) formulation is available, which should only be used to check the result of the first implementation in case of doubt.

$$\text{Minimize : } \sum_{v \in G} b_v$$

$$\text{Such that :}$$

$$\forall (u,v) \in G, d_u - d_v + n b_u + n b_v \geq 0$$
$$\forall u \in G, 0 \leq d_u \leq |G|$$

A brief explanation:

An acyclic digraph can be seen as a poset, and every poset has a linear extension. This means that in any acyclic digraph the vertices can be ordered with a total order $<$ in such a way that if $(u,v) \in G$, then $u < v$. Thus, this linear program is built in order to assign to each vertex v a number $d_v \in [0, \ldots, n-1]$ such that if there exists an edge $(u,v) \in G$ then either $d_v < d_u$ or one of u or v is removed. The number of vertices removed is then minimized, which is the objective.

EXAMPLES:

The necessary example:

```
sage: g = graphs.PetersenGraph()
sage: fvs = g.feedback_vertex_set()
sage: len(fvs)
3
sage: g.delete_vertices(fvs)
sage: g.is_forest()
True
```

In a digraph built from a graph, any edge is replaced by arcs going in the two opposite directions, thus creating a cycle of length two. Hence, to remove all the cycles from the graph, each edge must see one of its neighbors removed: a feedback vertex set is in this situation a vertex cover:

```
sage: cycle = graphs.CycleGraph(5)
sage: dcycle = DiGraph(cycle)
sage: cycle.vertex_cover(value_only=True)
3
sage: feedback = dcycle.feedback_vertex_set()
```

(continues on next page)

(continued from previous page)

```
sage: len(feedback)
3
sage: u,v = next(cycle.edge_iterator(labels=None))
sage: u in feedback or v in feedback
True
```

For a circuit, the minimum feedback arc set is clearly 1:

```
sage: circuit = digraphs.Circuit(5)
sage: circuit.feedback_vertex_set(value_only=True) == 1
True
```

flow(*x*, *y*, *value_only=True*, *integer=False*, *use_edge_labels=True*, *vertex_bound=False*, *algorithm=None*, *solver=None*, *verbose=0*)

Return a maximum flow in the graph from `x` to `y`.

The returned flow is represented by an optimal valuation of the edges. For more information, see the Wikipedia article Max_flow.

As an optimization problem, is can be expressed this way :

$$\text{Maximize : } \sum_{e \in G.edges()} w_e b_e$$

$$\text{Such that : } \forall v \in G, \sum_{(u,v) \in G.edges()} b_{(u,v)} \leq 1$$

$$\forall x \in G, b_x \text{ is a binary variable}$$

INPUT:

- `x` – source vertex
- `y` – sink vertex
- `value_only` – boolean (default: `True`); whether to return only the value of a maximal flow, or to also return a flow graph (a copy of the current graph, such that each edge has the flow using it as a label, the edges without flow being omitted)
- `integer` – boolean (default: `True`); whether to compute an optimal solution under the constraint that the flow going through an edge has to be an integer, or without this constraint
- `use_edge_labels` – boolean (default: `False`); whether to compute a maximum flow where each edge has a capacity defined by its label (if an edge has no label, capacity 1 is assumed), or to use default edge capacity of 1
- `vertex_bound` – boolean (default: `False`); when set to `True`, sets the maximum flow leaving a vertex different from x to 1 (useful for vertex connectivity parameters)
- `algorithm` – string (default: `None`); the algorithm to use among:
  - `"FF"`, a Python implementation of the Ford-Fulkerson algorithm (only available when `vertex_bound = False`)
  - `"LP"`, the flow problem is solved using Linear Programming
  - `"igraph"`, the `igraph` implementation of the Goldberg-Tarjan algorithm is used (only available when `igraph` is installed and `vertex_bound = False`)

  When `algorithm = None` (default), we use `LP` if `vertex_bound = True`, otherwise, we use `igraph` if it is available, `FF` if it is not available.

- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.

  Only useful when LP is used to solve the flow problem.

- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default (quiet).

  Only useful when LP is used to solve the flow problem.

Note: Even though the three different implementations are meant to return the same Flow values, they can not be expected to return the same Flow graphs.

Besides, the use of Linear Programming may possibly mean a (slight) numerical noise.

EXAMPLES:

Two basic applications of the flow method for the `PappusGraph` and the `ButterflyGraph` with parameter 2

```
sage: g=graphs.PappusGraph()
sage: int(g.flow(1,2))
3
```

```
sage: b=digraphs.ButterflyGraph(2)
sage: int(b.flow(('00', 1), ('00', 2)))
1
```

The flow method can be used to compute a matching in a bipartite graph by linking a source s to all the vertices of the first set and linking a sink t to all the vertices of the second set, then computing a maximum $s - t$ flow

```
sage: g = DiGraph()
sage: g.add_edges(('s', i) for i in range(4))
sage: g.add_edges((i, 4 + j) for i in range(4) for j in range(4))
sage: g.add_edges((4 + i, 't') for i in range(4))
sage: [cardinal, flow_graph] = g.flow('s', 't', integer=True, value_
↪only=False)
sage: flow_graph.delete_vertices(['s', 't'])
sage: flow_graph.size()
4
```

The undirected case:

```
sage: g = Graph()
sage: g.add_edges(('s', i) for i in range(4))
sage: g.add_edges((i, 4 + j) for i in range(4) for j in range(4))
sage: g.add_edges((4 + i, 't') for i in range(4))
sage: [cardinal, flow_graph] = g.flow('s', 't', integer=True, value_
↪only=False)
sage: flow_graph.delete_vertices(['s', 't'])
sage: flow_graph.size()
4
```

genus (*set_embedding=True*, *on_embedding=None*, *minimal=True*, *maximal=False*, *circular=None*, *ordered=True*)

Return the minimal genus of the graph.

The genus of a compact surface is the number of handles it has. The genus of a graph is the minimal genus of the surface it can be embedded into. It can be seen as a measure of non-planarity; a planar graph has genus zero.

Note: This function uses Euler's formula and thus it is necessary to consider only connected graphs.

INPUT:

- `set_embedding` – boolean (default: `True`); whether or not to store an embedding attribute of the computed (minimal) genus of the graph
- **`on_embedding` – two kinds of input are allowed (default:** `None`):
  - a dictionary representing a combinatorial embedding on which the genus should be computed. Note that this must be a valid embedding for the graph. The dictionary structure is given by: `vertex1: [neighbor1, neighbor2, neighbor3], vertex2: [neighbor]` where there is a key for each vertex in the graph and a (clockwise) ordered list of each vertex's neighbors as values. The value of `on_embedding` takes precedence over a stored `_embedding` attribute if `minimal` is set to `False`.
  - The value `True`, in order to indicate that the embedding stored as `_embedding` should be used (see examples).
- `minimal` – boolean (default: `True`); whether or not to compute the minimal genus of the graph (i.e., testing all embeddings). If minimal is `False`, then either `maximal` must be `True` or `on_embedding` must not be `None`. If `on_embedding` is not `None`, it will take priority over `minimal`. Similarly, if `maximal` is `True`, it will take priority over `minimal`.
- `maximal` – boolean (default: `False`); whether or not to compute the maximal genus of the graph (i.e., testing all embeddings). If `maximal` is `False`, then either `minimal` must be `True` or `on_embedding` must not be `None`. If `on_embedding` is not `None`, it will take priority over `maximal`. However, `maximal` takes priority over the default `minimal`.
- `circular` – list (default: `None`); if `circular` is a list of vertices, the method computes the genus preserving a planar embedding of the this list. If `circular` is defined, `on_embedding` is not a valid option.
- `ordered` – boolean (default: `True`); if `circular` is `True`, then whether or not the boundary order may be permuted (default is `True`, which means the boundary order is preserved)

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.genus() # tests for minimal genus by default
1
sage: g.genus(on_embedding=True, maximal=True) # on_embedding overrides␣
↪minimal and maximal arguments
1
sage: g.genus(maximal=True) # setting maximal to True overrides default␣
↪minimal=True
3
sage: g.genus(on_embedding=g.get_embedding()) # can also send a valid␣
↪combinatorial embedding dict
3
sage: (graphs.CubeGraph(3)).genus()
0
sage: K23 = graphs.CompleteBipartiteGraph(2,3)
```

(continues on next page)

(continued from previous page)

```
sage: K23.genus()
0
sage: K33 = graphs.CompleteBipartiteGraph(3,3)
sage: K33.genus()
1
```

Using the circular argument, we can compute the minimal genus preserving a planar, ordered boundary:

```
sage: cube = graphs.CubeGraph(2)
sage: cube.genus(circular=['01','10'])
0
sage: cube.is_circular_planar()
True
sage: cube.genus(circular=['01','10'])
0
sage: cube.genus(circular=['01','10'], on_embedding=True)
Traceback (most recent call last):
...
ValueError: on_embedding is not a valid option when circular is defined
sage: cube.genus(circular=['01','10'], maximal=True)
Traceback (most recent call last):
...
NotImplementedError: cannot compute the maximal genus of a genus respecting a␣
↪boundary
```

Note: not everything works for multigraphs, looped graphs or digraphs. But the minimal genus is ultimately computable for every connected graph – but the embedding we obtain for the simple graph can't be easily converted to an embedding of a non-simple graph. Also, the maximal genus of a multigraph does not trivially correspond to that of its simple graph:

```
sage: G = DiGraph({0: [0, 1, 1, 1], 1: [2, 2, 3, 3], 2: [1, 3, 3], 3: [0, 3]})
sage: G.genus()
Traceback (most recent call last):
...
NotImplementedError: cannot work with embeddings of non-simple graphs
sage: G.to_simple().genus()
0
sage: G.genus(set_embedding=False)
0
sage: G.genus(maximal=True, set_embedding=False)
Traceback (most recent call last):
...
NotImplementedError: cannot compute the maximal genus of a graph with loops␣
↪or multiple edges
```

We break graphs with cut vertices into their blocks, which greatly speeds up computation of minimal genus. This is not implemented for maximal genus:

```
sage: G = graphs.RandomBlockGraph(10, 5)
sage: G.genus()
10
```

get_embedding()

Return the attribute `_embedding` if it exists.

`_embedding` is a dictionary organized with vertex labels as keys and a list of each vertex's neighbors in clockwise order.

Error-checked to insure valid embedding is returned.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.genus()
1
sage: G.get_embedding()
{0: [1, 4, 5], 1: [0, 2, 6], 2: [1, 3, 7], 3: [2, 4, 8], 4: [0, 3, 9], 5: [0,␣
↪7, 8], 6: [1, 9, 8], 7: [2, 5, 9], 8: [3, 6, 5], 9: [4, 6, 7]}
```

get_pos (*dim=2*)

Return the position dictionary.

The position dictionary specifies the coordinates of each vertex.

INPUT:

- `dim` – integer (default: 2); whether to return the position dictionary in the plane (`dim == 2`) or in the 3-dimensional space

EXAMPLES:

By default, the position of a graph is None:

```
sage: G = Graph()
sage: G.get_pos()
sage: G.get_pos() is None
True
sage: P = G.plot(save_pos=True)
sage: G.get_pos()
{}
```

Some of the named graphs come with a pre-specified positioning:

```
sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: (0.0, 1.0),
 ...
 9: (0.475..., 0.154...)}
```

get_vertex (*vertex*)

Retrieve the object associated with a given vertex.

If no associated object is found, `None` is returned.

INPUT:

- `vertex` – the given vertex

EXAMPLES:

```
sage: d = {0: graphs.DodecahedralGraph(), 1: graphs.FlowerSnark(), 2: graphs.
↪MoebiusKantorGraph(), 3: graphs.PetersenGraph()}
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

get_vertices(*verts=None*)

Return a dictionary of the objects associated to each vertex.

INPUT:

- `verts` – iterable container of vertices

EXAMPLES:

```
sage: d = {0: graphs.DodecahedralGraph(), 1: graphs.FlowerSnark(), 2: graphs.
↪MoebiusKantorGraph(), 3: graphs.PetersenGraph()}
sage: T = graphs.TetrahedralGraph()
sage: T.set_vertices(d)
sage: T.get_vertices([1, 2])
{1: Flower Snark: Graph on 20 vertices,
 2: Moebius-Kantor Graph: Graph on 16 vertices}
```

girth(*certificate=False*)

Return the girth of the graph.

The girth is the length of the shortest cycle in the graph (directed cycle if the graph is directed). Graphs without (directed) cycles have infinite girth.

INPUT:

- `certificate` – boolean (default: `False`); whether to return `(g, c)`, where `g` is the girth and `c` is a list of vertices of a (directed) cycle of length `g` in the graph, thus providing a certificate that the girth is at most `g`, or `None` if `g` infinite

EXAMPLES:

```
sage: graphs.TetrahedralGraph().girth()
3
sage: graphs.CubeGraph(3).girth()
4
sage: graphs.PetersenGraph().girth(certificate=True)  # random
(5, [4, 3, 2, 1, 0])
sage: graphs.HeawoodGraph().girth()
6
sage: next(graphs.trees(9)).girth()
+Infinity
```

See also:

- *odd_girth()* – return the odd girth of the graph.

graphplot(***options*)

Return a *GraphPlot* object.

See *GraphPlot* for more details.

INPUT:

- `**options` – parameters for the *GraphPlot* constructor

EXAMPLES:

Creating a *GraphPlot* object uses the same options as *plot()*:

```
sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
```

(continues on next page)

(continued from previous page)

```
....:      (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = g.graphplot(edge_labels=True, color_by_label=True, edge_style=
↪'dashed')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

We can modify the *GraphPlot* object. Notice that the changes are cumulative:

```
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
sage: GP.set_vertices(talk=True)
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

graphviz_string(*labels='string'*, *vertex_labels=True*, *edge_labels=False*, *edge_color=None*, *edge_colors=None*, *edge_options=()*, *color_by_label=False*, *rankdir='down'*, *subgraph_clusters=[]*, ***options*)

Return a representation in the `dot` language.

The `dot` language is a text based format for graphs. It is used by the software suite `graphviz`. The specifications of the language are available on the web (see the reference [dotspec]).

INPUT:

- `labels` – string (default: `"string"`); either `"string"` or `"latex"`. If labels is `"string"`, latex commands are not interpreted. This option stands for both vertex labels and edge labels.
- `vertex_labels` – boolean (default: `True`); whether to add the labels on vertices
- `edge_labels` – boolean (default: `False`); whether to add the labels on edges
- `edge_color` – (default: `None`); specify a default color for the edges. The color could be one of
  - a name given as a string such as `"blue"` or `"orchid"`
  - a HSV sequence in a string such as `".52,.386,.22"`
  - an hexadecimal code such as `"#DA3305"`
  - a 3-tuple of floating point (to be interpreted as RGB tuple). In this case the 3-tuple is converted in hexadecimal code.
- `edge_colors` – dictionary (default: `None`); a dictionary whose keys are colors and values are list of edges. The list of edges need not to be complete in which case the default color is used. See the option `edge_color` for a description of valid color formats.
- `color_by_label` – a boolean or dictionary or function (default: `False`); whether to color each edge with a different color according to its label; the colors are chosen along a rainbow, unless they are specified by a function or dictionary mapping labels to colors; this option is incompatible with `edge_color` and `edge_colors`. See the option `edge_color` for a description of valid color formats.
- `edge_options` – a function (or tuple thereof) mapping edges to a dictionary of options for this edge
- `rankdir` – `'left'`, `'right'`, `'up'`, or `'down'` (default: `'down'`, for consistency with `graphviz`): the preferred ranking direction for acyclic layouts; see the `rankdir` option of `graphviz`.

- `subgraph_clusters` – a list of lists of vertices (default: `[]`); From [dotspec]: "If supported, the layout engine will do the layout so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle. Note that, for good and bad, cluster subgraphs are not part of the `dot` language, but solely a syntactic convention adhered to by certain of the layout engines."

EXAMPLES:

```
sage: G = Graph({0: {1: None, 2: None}, 1: {0: None, 2: None}, 2: {0: None,␣
↪1: None, 3: 'foo'}, 3: {2: 'foo'}}, sparse=True)
sage: print(G.graphviz_string(edge_labels=True))
graph {
  node_0  [label="0"];
  node_1  [label="1"];
  node_2  [label="2"];
  node_3  [label="3"];

  node_0 -- node_1;
  node_0 -- node_2;
  node_1 -- node_2;
  node_2 -- node_3 [label="foo"];
}
```

A variant, with the labels in latex, for post-processing with `dot2tex`:

```
sage: print(G.graphviz_string(edge_labels=True, labels="latex"))
graph {
  node [shape="plaintext"];
  node_0  [label=" ", texlbl="$0$"];
  node_1  [label=" ", texlbl="$1$"];
  node_2  [label=" ", texlbl="$2$"];
  node_3  [label=" ", texlbl="$3$"];

  node_0 -- node_1;
  node_0 -- node_2;
  node_1 -- node_2;
  node_2 -- node_3 [label=" ", texlbl="$\text{\texttt{foo}}$"];
}
```

Same, with a digraph and a color for edges:

```
sage: G = DiGraph({0: {1: None, 2: None}, 1: {2: None}, 2: {3: 'foo'}, 3: {}},
↪ sparse=True)
sage: print(G.graphviz_string(edge_color="red"))
digraph {
  node_0  [label="0"];
  node_1  [label="1"];
  node_2  [label="2"];
  node_3  [label="3"];

edge [color="red"];
  node_0 -> node_1;
  node_0 -> node_2;
  node_1 -> node_2;
  node_2 -> node_3;
}
```

A digraph using latex labels for vertices and edges:

```
sage: f(x) = -1 / x
sage: g(x) = 1 / (x + 1)
sage: G = DiGraph()
sage: G.add_edges((i, f(i), f) for i in (1, 2, 1/2, 1/4))
sage: G.add_edges((i, g(i), g) for i in (1, 2, 1/2, 1/4))
sage: print(G.graphviz_string(labels="latex", edge_labels=True))  # random
digraph {
  node [shape="plaintext"];
  node_10  [label=" ", texlbl="$1$"];
  node_11  [label=" ", texlbl="$2$"];
  node_3  [label=" ", texlbl="$-\frac{1}{2}$"];
  node_6  [label=" ", texlbl="$\frac{1}{2}$"];
  node_7  [label=" ", texlbl="$\frac{1}{2}$"];
  node_5  [label=" ", texlbl="$\frac{1}{3}$"];
  node_8  [label=" ", texlbl="$\frac{2}{3}$"];
  node_4  [label=" ", texlbl="$\frac{1}{4}$"];
  node_1  [label=" ", texlbl="$-2$"];
  node_9  [label=" ", texlbl="$\frac{4}{5}$"];
  node_0  [label=" ", texlbl="$-4$"];
  node_2  [label=" ", texlbl="$-1$"];

  node_10 -> node_2 [label=" ", texlbl="$x \ {\mapsto}\ -\frac{1}{x}$"];
  node_10 -> node_6 [label=" ", texlbl="$x \ {\mapsto}\ \frac{1}{x + 1}$"];
  node_11 -> node_3 [label=" ", texlbl="$x \ {\mapsto}\ -\frac{1}{x}$"];
  node_11 -> node_5 [label=" ", texlbl="$x \ {\mapsto}\ \frac{1}{x + 1}$"];
  node_7 -> node_1 [label=" ", texlbl="$x \ {\mapsto}\ -\frac{1}{x}$"];
  node_7 -> node_8 [label=" ", texlbl="$x \ {\mapsto}\ \frac{1}{x + 1}$"];
  node_4 -> node_0 [label=" ", texlbl="$x \ {\mapsto}\ -\frac{1}{x}$"];
  node_4 -> node_9 [label=" ", texlbl="$x \ {\mapsto}\ \frac{1}{x + 1}$"];
}

sage: print(G.graphviz_string(labels="latex", color_by_label=True))  # random
digraph {
  node [shape="plaintext"];
  node_10  [label=" ", texlbl="$1$"];
  node_11  [label=" ", texlbl="$2$"];
  node_3  [label=" ", texlbl="$-\frac{1}{2}$"];
  node_6  [label=" ", texlbl="$\frac{1}{2}$"];
  node_7  [label=" ", texlbl="$\frac{1}{2}$"];
  node_5  [label=" ", texlbl="$\frac{1}{3}$"];
  node_8  [label=" ", texlbl="$\frac{2}{3}$"];
  node_4  [label=" ", texlbl="$\frac{1}{4}$"];
  node_1  [label=" ", texlbl="$-2$"];
  node_9  [label=" ", texlbl="$\frac{4}{5}$"];
  node_0  [label=" ", texlbl="$-4$"];
  node_2  [label=" ", texlbl="$-1$"];

  node_10 -> node_2 [color = "#ff0000"];
  node_10 -> node_6 [color = "#00ffff"];
  node_11 -> node_3 [color = "#ff0000"];
  node_11 -> node_5 [color = "#00ffff"];
  node_7 -> node_1 [color = "#ff0000"];
  node_7 -> node_8 [color = "#00ffff"];
  node_4 -> node_0 [color = "#ff0000"];
  node_4 -> node_9 [color = "#00ffff"];
}
```

(continues on next page)

(continued from previous page)

```
sage: print(G.graphviz_string(labels="latex", color_by_label={f: "red", g:
↪"blue"}))  # random
digraph {
  node [shape="plaintext"];
  node_10  [label=" ", texlbl="$1$"];
  node_11  [label=" ", texlbl="$2$"];
  node_3  [label=" ", texlbl="$-\frac{1}{2}$"];
  node_6  [label=" ", texlbl="$\frac{1}{2}$"];
  node_7  [label=" ", texlbl="$\frac{1}{2}$"];
  node_5  [label=" ", texlbl="$\frac{1}{3}$"];
  node_8  [label=" ", texlbl="$\frac{2}{3}$"];
  node_4  [label=" ", texlbl="$\frac{1}{4}$"];
  node_1  [label=" ", texlbl="$-2$"];
  node_9  [label=" ", texlbl="$\frac{4}{5}$"];
  node_0  [label=" ", texlbl="$-4$"];
  node_2  [label=" ", texlbl="$-1$"];

  node_10 -> node_2 [color = "red"];
  node_10 -> node_6 [color = "blue"];
  node_11 -> node_3 [color = "red"];
  node_11 -> node_5 [color = "blue"];
  node_7 -> node_1 [color = "red"];
  node_7 -> node_8 [color = "blue"];
  node_4 -> node_0 [color = "red"];
  node_4 -> node_9 [color = "blue"];
}
```

By default `graphviz` renders digraphs using a hierarchical layout, ranking the vertices down from top to bottom. Here we specify alternative ranking directions for this layout:

```
sage: D = DiGraph([(1, 2)])
sage: print(D.graphviz_string(rankdir="up"))
digraph {
  rankdir=BT
  node_0  [label="1"];
  node_1  [label="2"];

  node_0 -> node_1;
}
sage: print(D.graphviz_string(rankdir="down"))
digraph {
  node_0  [label="1"];
  node_1  [label="2"];

  node_0 -> node_1;
}
sage: print(D.graphviz_string(rankdir="left"))
digraph {
  rankdir=RL
  node_0  [label="1"];
  node_1  [label="2"];

  node_0 -> node_1;
}
sage: print(D.graphviz_string(rankdir="right"))
digraph {
```

(continues on next page)

(continued from previous page)

```
  rankdir=LR
  node_0  [label="1"];
  node_1  [label="2"];

  node_0 -> node_1;
}
```

Edge-specific options can also be specified by providing a function (or tuple thereof) which maps each edge to a dictionary of options. Valid options are `"color"`, `"backward"` (a boolean), `"dot"` (a string containing a sequence of options in `dot` format), `"label"` (a string), `"label_style"` (`"string"` or `"latex"`), `"edge_string"` (`"--"` or `"->"`). Here we state that the graph should be laid out so that edges starting from `1` are going backward (e.g. going up instead of down):

```
sage: def edge_options(data):
....:     u, v, label = data
....:     return {"backward": u == 1}
sage: print(G.graphviz_string(edge_options=edge_options))  # random
digraph {
  node_10  [label="1"];
  node_11  [label="2"];
  node_3  [label="-1/2"];
  node_6  [label="1/2"];
  node_7  [label="1/2"];
  node_5  [label="1/3"];
  node_8  [label="2/3"];
  node_4  [label="1/4"];
  node_1  [label="-2"];
  node_9  [label="4/5"];
  node_0  [label="-4"];
  node_2  [label="-1"];

  node_2 -> node_10 [dir=back];
  node_6 -> node_10 [dir=back];
  node_11 -> node_3;
  node_11 -> node_5;
  node_7 -> node_1;
  node_7 -> node_8;
  node_4 -> node_0;
  node_4 -> node_9;
}
```

We now test all options:

```
sage: def edge_options(data):
....:     u, v, label = data
....:     options = {"color": {f: "red", g: "blue"}[label]}
....:     if (u,v) == (1/2, -2): options["label"]       = "coucou"; options[
↪"label_style"] = "string"
....:     if (u,v) == (1/2,2/3): options["dot"]         = "x=1,y=2"
....:     if (u,v) == (1,   -1): options["label_style"] = "latex"
....:     if (u,v) == (1,  1/2): options["edge_string"] = "<-"
....:     if (u,v) == (1/2,  1): options["backward"]    = True
....:     return options
sage: print(G.graphviz_string(edge_options=edge_options))  # random
digraph {
  node_10  [label="1"];
```

(continues on next page)

(continued from previous page)

```
  node_11  [label="2"];
  node_3  [label="-1/2"];
  node_6  [label="1/2"];
  node_7  [label="1/2"];
  node_5  [label="1/3"];
  node_8  [label="2/3"];
  node_4  [label="1/4"];
  node_1  [label="-2"];
  node_9  [label="4/5"];
  node_0  [label="-4"];
  node_2  [label="-1"];

  node_10 -> node_2 [label=" ", texlbl="$x \ {\mapsto}\ -\frac{1}{x}$", color␣
↪= "red"];
  node_10 <- node_6 [color = "blue"];
  node_11 -> node_3 [color = "red"];
  node_11 -> node_5 [color = "blue"];
  node_7 -> node_1 [label="coucou", color = "red"];
  node_7 -> node_8 [x=1,y=2, color = "blue"];
  node_4 -> node_0 [color = "red"];
  node_4 -> node_9 [color = "blue"];
}
```

graphviz_to_file_named(*filename*, ***options*)

Write a representation in the `dot` language in a file.

The `dot` language is a plaintext format for graph structures. See the documentation of *graphviz_string()* for available options.

INPUT:

- `filename` – the name of the file to write in
- `**options` – options for the graphviz string

EXAMPLES:

```
sage: G = Graph({0: {1: None, 2: None}, 1: {0: None, 2: None}, 2: {0: None,␣
↪1: None, 3: 'foo'}, 3: {2: 'foo'}}, sparse=True)
sage: tempfile = os.path.join(SAGE_TMP, 'temp_graphviz')
sage: G.graphviz_to_file_named(tempfile, edge_labels=True)
sage: with open(tempfile) as f:
....:     print(f.read())
graph {
  node_0  [label="0"];
  node_1  [label="1"];
  node_2  [label="2"];
  node_3  [label="3"];

  node_0 -- node_1;
  node_0 -- node_2;
  node_1 -- node_2;
  node_2 -- node_3 [label="foo"];
}
```

hamiltonian_cycle(*algorithm='tsp'*, *solver=None*, *constraint_generation=None*, *verbose=0*, *verbose_constraints=False*)

Return a Hamiltonian cycle/circuit of the current graph/digraph.

A graph (resp. digraph) is said to be Hamiltonian if it contains as a subgraph a cycle (resp. a circuit) going through all the vertices.

Computing a Hamiltonian cycle/circuit being NP-Complete, this algorithm could run for some time depending on the instance.

ALGORITHM:

See *traveling_salesman_problem()* for 'tsp' algorithm and *find_hamiltonian()* from *sage.graphs.generic_graph_pyx* for 'backtrack' algorithm.

INPUT:

- `algorithm` – string (default: `'tsp'`); one of 'tsp' or 'backtrack'
- `solver` – (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `constraint_generation` – boolean (default: `None`); whether to use constraint generation when solving the Mixed Integer Linear Program.

  When `constraint_generation = None`, constraint generation is used whenever the graph has a density larger than 70%.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.
- `verbose_constraints` – boolean (default: `False`); whether to display which constraints are being generated

OUTPUT:

If using the `'tsp'` algorithm, returns a Hamiltonian cycle/circuit if it exists; otherwise, raises a `EmptySetError` exception. If using the `'backtrack'` algorithm, returns a pair `(B, P)`. If `B` is `True` then `P` is a Hamiltonian cycle and if `B` is `False`, `P` is a longest path found by the algorithm. Observe that if `B` is `False`, the graph may still be Hamiltonian. The `'backtrack'` algorithm is only implemented for undirected graphs.

Warning: The `'backtrack'` algorithm may loop endlessly on graphs with vertices of degree 1.

NOTE:

This function, as *is_hamiltonian()*, computes a Hamiltonian cycle if it exists: the user should *NOT* test for Hamiltonicity using *is_hamiltonian()* before calling this function, as it would result in computing it twice.

The backtrack algorithm is only implemented for undirected graphs.

EXAMPLES:

The Heawood Graph is known to be Hamiltonian

```
sage: g = graphs.HeawoodGraph()
sage: g.hamiltonian_cycle()
TSP from Heawood graph: Graph on 14 vertices
```

The Petersen Graph, though, is not

```
sage: g = graphs.PetersenGraph()
sage: g.hamiltonian_cycle()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
EmptySetError: the given graph is not Hamiltonian
```

Now, using the backtrack algorithm in the Heawood graph

```
sage: G=graphs.HeawoodGraph()
sage: G.hamiltonian_cycle(algorithm='backtrack')
(True, [11, 10, 1, 2, 3, 4, 9, 8, 7, 6, 5, 0, 13, 12])
```

And now in the Petersen graph

```
sage: G=graphs.PetersenGraph()
sage: G.hamiltonian_cycle(algorithm='backtrack')
(False, [6, 8, 5, 0, 1, 2, 7, 9, 4, 3])
```

Finally, we test the algorithm in a cube graph, which is Hamiltonian

```
sage: G=graphs.CubeGraph(3)
sage: G.hamiltonian_cycle(algorithm='backtrack')
(True, ['010', '110', '100', '000', '001', '101', '111', '011'])
```

hamiltonian_path(*s=None*, *t=None*, *use_edge_labels=False*, *maximize=False*, *algorithm='MILP'*, *solver=None*, *verbose=0*)

Return a Hamiltonian path of the current graph/digraph.

A path is Hamiltonian if it goes through all the vertices exactly once. Computing a Hamiltonian path being NP-Complete, this algorithm could run for some time depending on the instance.

When `use_edge_labels == True`, this method returns either a minimum weight hamiltonian path or a maximum weight Hamiltonian path (if `maximize == True`).

See also:

- *longest_path()*
- *hamiltonian_cycle()*

INPUT:

- `s` – vertex (default: `None`); if specified, then forces the source of the path (the method then returns a Hamiltonian path starting at `s`)
- `t` – vertex (default: `None`); if specified, then forces the destination of the path (the method then returns a Hamiltonian path ending at `t`)
- `use_edge_labels` – boolean (default: `False`); whether to compute a weighted hamiltonian path where the weight of an edge is defined by its label (a label set to `None` or `{}` being considered as a weight of 1), or a non-weighted hamiltonian path
- `maximize` – boolean (default: `False`); whether to compute a minimum (default) or a maximum (when `maximize == True`) weight hamiltonian path. This parameter is considered only if `use_edge_labels == True`.
- `algorithm` – string (default: `"MILP"`); the algorithm the use among `"MILP"` and `"backtrack"`; two remarks on this respect:
  - While the MILP formulation returns an exact answer, the backtrack algorithm is a randomized heuristic.

  - The backtrack algorithm does not support edge weighting, so setting `use_edge_labels=True` will force the use of the MILP algorithm.

- `solver` – string (default: `None`); specifies the Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve`
- `verbose` – integer (default: `0`); sets the level of verbosity with 0 meaning quiet

OUTPUT:

A subgraph of `self` corresponding to a (directed if `self` is directed) hamiltonian path. If no hamiltonian path is found, return `None`. If `use_edge_labels == True`, a pair `weight, path` is returned.

EXAMPLES:

The 3×3-grid has an Hamiltonian path, an hamiltonian path starting from vertex $(0, 0)$ and ending at vertex $(2, 2)$, but no Hamiltonian path starting from $(0, 0)$ and ending at $(0, 1)$:

```
sage: g = graphs.Grid2dGraph(3, 3)
sage: g.hamiltonian_path()
Hamiltonian path from 2D Grid Graph for [3, 3]: Graph on 9 vertices
sage: g.hamiltonian_path(s=(0, 0), t=(2, 2))
Hamiltonian path from 2D Grid Graph for [3, 3]: Graph on 9 vertices
sage: g.hamiltonian_path(s=(0, 0), t=(2, 2), use_edge_labels=True)
(8, Hamiltonian path from 2D Grid Graph for [3, 3]: Graph on 9 vertices)
sage: g.hamiltonian_path(s=(0, 0), t=(0, 1)) is None
True
sage: g.hamiltonian_path(s=(0, 0), t=(0, 1), use_edge_labels=True)
(0, None)
```

has_edge(*u*, *v=None*, *label=None*)

Check whether `(u, v)` is an edge of the (di)graph.

INPUT: The following forms are accepted:

- G.has_edge(1, 2)
- G.has_edge((1, 2))
- G.has_edge(1, 2, 'label')
- G.has_edge((1, 2, 'label'))

EXAMPLES:

```
sage: graphs.EmptyGraph().has_edge(9, 2)
False
sage: DiGraph().has_edge(9, 2)
False
sage: G = Graph(sparse=True)
sage: G.add_edge(0, 1, "label")
sage: G.has_edge(0, 1, "different label")
False
sage: G.has_edge(0, 1, "label")
True
```

has_loops()

Return whether there are loops in the (di)graph

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0, 0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0, 0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

has_multiple_edges(*to_undirected=False*)

Return whether there are multiple edges in the (di)graph.

INPUT:

- `to_undirected` – (default: `False`); if `True`, runs the test on the undirected version of a DiGraph. Otherwise, treats DiGraph edges `(u, v)` and `(v, u)` as unique individual edges.

EXAMPLES:

```
sage: G = Graph(multiedges=True, sparse=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0, 1)] * 3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
```

(continues on next page)

(continued from previous page)

```
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True, sparse=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0, 1)] * 3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]

sage: G = DiGraph({1: {2: 'h'}, 2: {1: 'g'}}, sparse=True)
sage: G.has_multiple_edges()
False
sage: G.has_multiple_edges(to_undirected=True)
True
sage: G.multiple_edges()
[]
sage: G.multiple_edges(to_undirected=True)
[(1, 2, 'h'), (2, 1, 'g')]
```

A loop is not a multiedge:

```
sage: g = Graph(loops=True, multiedges=True)
sage: g.add_edge(0, 0)
sage: g.has_multiple_edges()
False
```

has_vertex(*vertex*)

Check if `vertex` is one of the vertices of this graph.

INPUT:

- `vertex` – the name of a vertex (see *add_vertex()*)

EXAMPLES:

```
sage: g = Graph({0: [1, 2, 3], 2: [4]}); g
Graph on 5 vertices
sage: 2 in g
True
sage: 10 in g
False
sage: graphs.PetersenGraph().has_vertex(99)
False
```

igraph_graph(*vertex_list=None*, *vertex_attrs={}*, *edge_attrs={}*)

Return an `igraph` graph from the Sage graph.

Optionally, it is possible to add vertex attributes and edge attributes to the output graph.

Note: This routine needs the optional package igraph to be installed: to do so, it is enough to run `sage -i python_igraph`. For more information on the Python version of igraph, see http://igraph.org/python/.

INPUT:

- `vertex_list` – list (default: `None`); defines a mapping from the vertices of the graph to consecutive integers in `(0, \ldots, n-1)`. Otherwise, the result of `:meth:`vertices` will be used instead. Because `:meth:`vertices` only works if the vertices can be sorted, using ``vertex_list` is useful when working with possibly non-sortable objects in Python 3.
- `vertex_attrs` – dictionary (default: `{}`); a dictionary where the key is a string (the attribute name), and the value is an iterable containing in position i the label of the i-th vertex in the list `vertex_list` if it is given or in *`vertices()`* when `vertex_list == None` (see http://igraph.org/python/doc/igraph.Graph-class.html#__init__ for more information)
- `edge_attrs` – dictionary (default: `{}`); a dictionary where the key is a string (the attribute name), and the value is an iterable containing in position i the label of the i-th edge in the list outputted by *`edge_iterator()`* (see http://igraph.org/python/doc/igraph.Graph-class.html#__init__ for more information)

Note: In `igraph`, a graph is weighted if the edge labels have attribute `weight`. Hence, to create a weighted graph, it is enough to add this attribute.

Note: Often, Sage uses its own defined types for integer/floats. These types may not be igraph-compatible (see example below).

EXAMPLES:

Standard conversion:

```
sage: G = graphs.TetrahedralGraph()
sage: H = G.igraph_graph()           # optional - python_igraph
sage: H.summary()                    # optional - python_igraph
'IGRAPH U--- 4 6 -- '
sage: G = digraphs.Path(3)
sage: H = G.igraph_graph()           # optional - python_igraph
sage: H.summary()                    # optional - python_igraph
'IGRAPH D--- 3 2 -- '
```

Adding edge attributes:

```
sage: G = Graph([(1, 2, 'a'), (2, 3, 'b')])
sage: E = list(G.edge_iterator())
sage: H = G.igraph_graph(edge_attrs={'label': [e[2] for e in E]}) # optional -
↪ python_igraph
sage: H.es['label']                                               # optional -
↪ python_igraph
```

(continues on next page)

(continued from previous page)

```
['a', 'b']
```

If edges have an attribute `weight`, the igraph graph is considered weighted:

```
sage: G = Graph([(1, 2, {'weight': 1}), (2, 3, {'weight': 2})])
sage: E = list(G.edge_iterator())
sage: H = G.igraph_graph(edge_attrs={'weight': [e[2]['weight'] for e in E]})
↪# optional - python_igraph
sage: H.is_weighted()
↪# optional - python_igraph
True
sage: H.es['weight']
↪# optional - python_igraph
[1, 2]
```

Adding vertex attributes:

```
sage: G = graphs.GridGraph([2, 2])
sage: H = G.igraph_graph(vertex_attrs={'name': G.vertices()}) # optional -␣
↪python_igraph
sage: H.vs()['name']                                          # optional -␣
↪python_igraph
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

Providing a mapping from vertices to consecutive integers:

```
sage: G = graphs.GridGraph([2, 2])
sage: V = list(G)
sage: H = G.igraph_graph(vertex_list=V, vertex_attrs={'name': V}) # optional -
↪ python_igraph
sage: H.vs()['name'] == V                                         # optional -
↪ python_igraph
True
```

Sometimes, Sage integer/floats are not compatible with igraph:

```
sage: G = Graph([(0, 1, 2)])
sage: E = list(G.edge_iterator())
sage: H = G.igraph_graph(edge_attrs={'capacity': [e[2] for e in E]}) #␣
↪optional - python_igraph
sage: H.maxflow_value(0, 1, 'capacity')                              #␣
↪optional - python_igraph
1.0
sage: H = G.igraph_graph(edge_attrs={'capacity': [float(e[2]) for e in E]}) #␣
↪optional - python_igraph
sage: H.maxflow_value(0, 1, 'capacity')                                    #␣
↪optional - python_igraph
2.0
```

incidence_matrix(*oriented=None*, *sparse=True*, *vertices=None*, *edges=None*)

Return the incidence matrix of the (di)graph.

Each row is a vertex, and each column is an edge. The vertices are ordered as obtained by the method *vertices()*, except when parameter `vertices` is given (see below), and the edges as obtained by the method *edge_iterator()*.

If the graph is not directed, then return a matrix with entries in $\{0, 1, 2\}$. Each column will either contain two 1 (at the position of the endpoint of the edge), or one 2 (if the corresponding edge is a loop).

If the graph is directed return a matrix in $\{-1, 0, 1\}$ where -1 and $+1$ correspond respectively to the source and the target of the edge. A loop will correspond to a zero column. In particular, it is not possible to recover the loops of an oriented graph from its incidence matrix.

See the Wikipedia article Incidence_matrix for more information.

INPUT:

- `oriented` – boolean (default: `None`); when set to `True`, the matrix will be oriented (i.e. with entries in -1, 0, 1) and if set to `False` the matrix will be not oriented (i.e. with entries in 0, 1, 2). By default, this argument is inferred from the graph type. Note that in the case the graph is not directed and with the option `directed=True`, a somewhat random direction is chosen for each edge.
- `sparse` – boolean (default: `True`); whether to use a sparse or a dense matrix
- `vertices` – list (default: `None`); when specified, the i-th row of the matrix corresponds to the i-th vertex in the ordering of `vertices`, otherwise, the i-th row of the matrix corresponds to the i-th vertex in the ordering given by method *`vertices()`*.
- `edges` – list (default: `None`); when specified, the i-th column of the matrix corresponds to the i-th edge in the ordering of `edges`, otherwise, the i-th column of the matrix corresponds to the i-th edge in the ordering given by method *`edge_iterator()`*.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.incidence_matrix()
[1 1 1 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 1 1 0 0 0 0 0]
[0 1 0 0 0 0 0 1 0 1 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 1 1 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 1 0 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 1 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 1]
sage: G.incidence_matrix(oriented=True)
[-1 -1 -1  0  0  0  0  0  0  0  0  0  0  0]
[ 1  0  0 -1 -1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0 -1 -1  0  0  0  0  0  0  0]
[ 0  0  0  0  0  1  0 -1 -1  0  0  0  0  0]
[ 0  1  0  0  0  0  0  1  0 -1  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0 -1 -1  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0  0 -1 -1]
[ 0  0  0  0  0  0  1  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0  1  1  0]
[ 0  0  0  0  0  0  0  0  0  1  0  0  0  1]

sage: G = digraphs.Circulant(4, [1, 3])
sage: G.incidence_matrix()
[-1 -1  1  0  0  0  1  0]
[ 1  0 -1 -1  1  0  0  0]
[ 0  0  0  1 -1 -1  0  1]
[ 0  1  0  0  0  1 -1 -1]

sage: graphs.CompleteGraph(3).incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
```

(continues on next page)

(continued from previous page)

```
sage: G = Graph([(0, 0), (0, 1), (0, 1)], loops=True, multiedges=True)
sage: G.incidence_matrix(oriented=False)
[2 1 1]
[0 1 1]
```

A well known result states that the product of the (oriented) incidence matrix with its transpose of a (non-oriented graph) is in fact the Kirchhoff matrix:

```
sage: G = graphs.PetersenGraph()
sage: m = G.incidence_matrix(oriented=True)
sage: m * m.transpose() == G.kirchhoff_matrix()
True

sage: K = graphs.CompleteGraph(3)
sage: m = K.incidence_matrix(oriented=True)
sage: m * m.transpose() == K.kirchhoff_matrix()
True

sage: H = Graph([(0, 0), (0, 1), (0, 1)], loops=True, multiedges=True)
sage: m = H.incidence_matrix(oriented=True)
sage: m * m.transpose() == H.kirchhoff_matrix()
True
```

A different ordering of the vertices:

```
sage: P5 = graphs.PathGraph(5)
sage: P5.incidence_matrix()
[1 0 0 0]
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
[0 0 0 1]
sage: P5.incidence_matrix(vertices=[2, 4, 1, 3, 0])
[0 1 1 0]
[0 0 0 1]
[1 1 0 0]
[0 0 1 1]
[1 0 0 0]
```

A different ordering of the edges:

```
sage: E = list(P5.edge_iterator(labels=False))
sage: P5.incidence_matrix(edges=E[::-1])
[0 0 0 1]
[0 0 1 1]
[0 1 1 0]
[1 1 0 0]
[1 0 0 0]
sage: P5.incidence_matrix(vertices=[2, 4, 1, 3, 0], edges=E[::-1])
[0 1 1 0]
[1 0 0 0]
[0 0 1 1]
[1 1 0 0]
[0 0 0 1]
```

is_bipartite(*certificate=False*)

Check whether the graph is bipartite.

Traverse the graph G with breadth-first-search and color nodes.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate. If set to `True`, the certificate returned is a proper 2-coloring when G is bipartite, and an odd cycle otherwise.

EXAMPLES:

```
sage: graphs.CycleGraph(4).is_bipartite()
True
sage: graphs.CycleGraph(5).is_bipartite()
False
sage: graphs.RandomBipartite(10, 10, 0.7).is_bipartite()
True
```

A random graph is very rarely bipartite:

```
sage: g = graphs.PetersenGraph()
sage: g.is_bipartite()
False
sage: false, oddcycle = g.is_bipartite(certificate=True)
sage: len(oddcycle) % 2
1
```

The method works identically with oriented graphs:

```
sage: g = DiGraph({0: [1, 2, 3], 2: [1], 3: [4]})
sage: g.is_bipartite()
False
sage: false, oddcycle = g.is_bipartite(certificate=True)
sage: len(oddcycle) % 2
1

sage: graphs.CycleGraph(4).random_orientation().is_bipartite()
True
sage: graphs.CycleGraph(5).random_orientation().is_bipartite()
False
```

is_cayley(*return_group=False*, *mapping=False*, *generators=False*, *allow_disconnected=False*)

Check whether the graph is a Cayley graph.

If none of the parameters are `True`, return a boolean indicating whether the graph is a Cayley graph. Otherwise, return a tuple containing said boolean and the requested data. If the graph is not a Cayley graph, each of the data will be `None`.

The empty graph is defined to be not a Cayley graph.

Note: For this routine to work on all graphs, the optional package `gap_packages` needs to be installed: to do so, it is enough to run `sage -i gap_packages`.

INPUT:

- `return_group` (boolean; `False`) – If True, return a group for which the graph is a Cayley graph.
- `mapping` (boolean; `False`) – If True, return a mapping from vertices to group elements.
- `generators` (boolean; `False`) – If True, return the generating set of the Cayley graph.

- `allow_disconnected` (boolean; `False`) – If True, disconnected graphs are considered Cayley if they can be obtained from the Cayley construction with a generating set that does not generate the group.

ALGORITHM:

For connected graphs, find a regular subgroup of the automorphism group. For disconnected graphs, check that the graph is vertex-transitive and perform the check on one of its connected components. If a simple graph has density over 1/2, perform the check on its complement as its disconnectedness may increase performance.

EXAMPLES:

A Petersen Graph is not a Cayley graph:

```
sage: g = graphs.PetersenGraph()
sage: g.is_cayley()
False
```

A Cayley digraph is a Cayley graph:

```
sage: C7 = groups.permutation.Cyclic(7)
sage: S = [(1,2,3,4,5,6,7), (1,3,5,7,2,4,6), (1,5,2,6,3,7,4)]
sage: d = C7.cayley_graph(generators=S)
sage: d.is_cayley()
True
```

Graphs with loops and multiedges will have identity and repeated elements, respectively, among the generators:

```
sage: g = Graph(graphs.PaleyGraph(9), loops=True, multiedges=True)
sage: g.add_edges([(u, u) for u in g])
sage: g.add_edges([(u, u+1) for u in g])
sage: _, S = g.is_cayley(generators=True)
sage: S # random
[(),
 (0,2,1)(a,a + 2,a + 1)(2*a,2*a + 2,2*a + 1),
 (0,2,1)(a,a + 2,a + 1)(2*a,2*a + 2,2*a + 1),
 (0,1,2)(a,a + 1,a + 2)(2*a,2*a + 1,2*a + 2),
 (0,1,2)(a,a + 1,a + 2)(2*a,2*a + 1,2*a + 2),
 (0,2*a + 2,a + 1)(1,2*a,a + 2)(2,2*a + 1,a),
 (0,a + 1,2*a + 2)(1,a + 2,2*a)(2,a,2*a + 1)]
```

is_chordal(*certificate=False*, *algorithm='B'*)

Check whether the given graph is chordal.

A Graph G is said to be chordal if it contains no induced hole (a cycle of length at least 4).

Alternatively, chordality can be defined using a Perfect Elimination Order :

A Perfect Elimination Order of a graph G is an ordering $v_1, ..., v_n$ of its vertex set such that for all i, the neighbors of v_i whose index is greater that i induce a complete subgraph in G. Hence, the graph G can be totally erased by successively removing vertices whose neighborhood is a clique (also called *simplicial* vertices) [FG1965].

(It can be seen that if G contains an induced hole, then it can not have a perfect elimination order. Indeed, if we write $h_1, ..., h_k$ the k vertices of such a hole, then the first of those vertices to be removed would have two non-adjacent neighbors in the graph.)

A Graph is then chordal if and only if it has a Perfect Elimination Order.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate.
  - If `certificate = False` (default), returns `True` or `False` accordingly.
  - If `certificate = True`, returns :
    * `(True, peo)` when the graph is chordal, where `peo` is a perfect elimination order of its vertices.
    * `(False, Hole)` when the graph is not chordal, where `Hole` (a `Graph` object) is an induced subgraph of `self` isomorphic to a hole.
- `algorithm` – string (default: `"B"`); the algorithm to choose among `"A"` or `"B"` (see next section). While they will agree on whether the given graph is chordal, they can not be expected to return the same certificates.

ALGORITHM:

This algorithm works through computing a Lex BFS on the graph, then checking whether the order is a Perfect Elimination Order by computing for each vertex v the subgraph induces by its non-deleted neighbors, then testing whether this graph is complete.

This problem can be solved in $O(m)$ [RT1975] (where m is the number of edges in the graph) but this implementation is not linear because of the complexity of Lex BFS.

EXAMPLES:

The lexicographic product of a Path and a Complete Graph is chordal

```
sage: g = graphs.PathGraph(5).lexicographic_product(graphs.CompleteGraph(3))
sage: g.is_chordal()
True
```

The same goes with the product of a random lobster (which is a tree) and a Complete Graph

```
sage: g = graphs.RandomLobster(10, .5, .5).lexicographic_product(graphs.
↪CompleteGraph(3))
sage: g.is_chordal()
True
```

The disjoint union of chordal graphs is still chordal:

```
sage: (2 * g).is_chordal()
True
```

Let us check the certificate given by Sage is indeed a perfect elimination order:

```
sage: _, peo = g.is_chordal(certificate=True)
sage: for v in peo:
....:     if not g.subgraph(g.neighbors(v)).is_clique():
....:         raise ValueError("this should never happen")
....:     g.delete_vertex(v)
```

Of course, the Petersen Graph is not chordal as it has girth 5:

```
sage: g = graphs.PetersenGraph()
sage: g.girth()
5
sage: g.is_chordal()
False
```

We can even obtain such a cycle as a certificate:

```
sage: _, hole = g.is_chordal(certificate=True)
sage: hole
Subgraph of (Petersen graph): Graph on 5 vertices
sage: hole.is_isomorphic(graphs.CycleGraph(5))
True
```

is_circulant(*certificate=False*)

Check whether the graph is circulant.

For more information, see Wikipedia article Circulant_graph.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate for yes-answers (see OUTPUT section)

OUTPUT:

When `certificate` is set to `False` (default) this method only returns `True` or `False` answers. When `certificate` is set to `True`, the method either returns `(False, None)` or `(True, lists_of_parameters)` each element of `lists_of_parameters` can be used to define the graph as a circulant graph.

See the documentation of *CirculantGraph()* and *Circulant()* for more information, and the examples below.

See also:

CirculantGraph() – a constructor for circulant graphs.

EXAMPLES:

The Petersen graph is not a circulant graph:

```
sage: g = graphs.PetersenGraph()
sage: g.is_circulant()
False
```

A cycle is obviously a circulant graph, but several sets of parameters can be used to define it:

```
sage: g = graphs.CycleGraph(5)
sage: g.is_circulant(certificate=True)
(True, [(5, [1, 4]), (5, [2, 3])])
```

The same goes for directed graphs:

```
sage: g = digraphs.Circuit(5)
sage: g.is_circulant(certificate=True)
(True, [(5, [1]), (5, [3]), (5, [2]), (5, [4])])
```

With this information, it is very easy to create (and plot) all possible drawings of a circulant graph:

```
sage: g = graphs.CirculantGraph(13, [2, 3, 10, 11])
sage: for param in g.is_circulant(certificate=True)[1]:
....:     graphs.CirculantGraph(*param)
Circulant graph ([2, 3, 10, 11]): Graph on 13 vertices
Circulant graph ([1, 5, 8, 12]): Graph on 13 vertices
Circulant graph ([4, 6, 7, 9]): Graph on 13 vertices
```

is_circular_planar(*on_embedding=None*, *kuratowski=False*, *set_embedding=True*, *boundary=None*, *ordered=False*, *set_pos=False*)

Check whether the graph is circular planar (outerplanar)

A graph is circular planar if it has a planar embedding in which all vertices can be drawn in order on a circle. This method can also be used to check the existence of a planar embedding in which the vertices of a specific set (the *boundary*) can be drawn on a circle, all other vertices being drawn inside of the circle. An order can be defined on the vertices of the boundary in order to define how they are to appear on the circle.

INPUT:

- `on_embedding` – dictionary (default: `None`); the embedding dictionary to test planarity on (i.e.: will return `True` or `False` only for the given embedding)
- `kuratowski` – boolean (default: `False`); whether to return a tuple with boolean first entry and the Kuratowski subgraph (i.e. an edge subdivision of K_5 or $K_{3,3}$) as the second entry (see OUTPUT below)
- `set_embedding` – boolean (default: `True`); whether or not to set the instance field variable that contains a combinatorial embedding (clockwise ordering of neighbors at each vertex). This value will only be set if a circular planar embedding is found. It is stored as a Python dict: `v1: [n1,n2,n3]` where `v1` is a vertex and `n1,n2,n3` are its neighbors.
- `boundary` – list (default: `None`); an ordered list of vertices that are required to be drawn on the circle, all others being drawn inside of it. It is set to `None` by default, meaning that *all* vertices should be drawn on the boundary.
- `ordered` – boolean (default: `False`); whether or not to consider the order of the boundary. It required `boundary` to be defined.
- `set_pos` – boolean (default: `False`); whether or not to set the position dictionary (for plotting) to reflect the combinatorial embedding. Note that this value will default to `False` if `set_embedding` is set to `False`. Also, the position dictionary will only be updated if a circular planar embedding is found.

OUTPUT:

The method returns `True` if the graph is circular planar, and `False` if it is not.

If `kuratowski` is set to `True`, then this function will return a tuple, whose first entry is a boolean and whose second entry is the Kuratowski subgraph (i.e. an edge subdivision of K_5 or $K_{3,3}$) isolated by the Boyer-Myrvold algorithm. Note that this graph might contain a vertex or edges that were not in the initial graph. These would be elements referred to below as parts of the wheel and the star, which were added to the graph to require that the boundary can be drawn on the boundary of a disc, with all other vertices drawn inside (and no edge crossings).

ALGORITHM:

This is a linear time algorithm to test for circular planarity. It relies on the edge-addition planarity algorithm due to Boyer-Myrvold. We accomplish linear time for circular planarity by modifying the graph before running the general planarity algorithm.

REFERENCE:

[BM2004]

EXAMPLES:

```
sage: g439 = Graph({1: [5, 7], 2: [5, 6], 3: [6, 7], 4: [5, 6, 7]})
sage: g439.show()
sage: g439.is_circular_planar(boundary=[1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
False
sage: g439.is_circular_planar(kuratowski=True, boundary=[1, 2, 3, 4])
(False, Graph on 8 vertices)
sage: g439.is_circular_planar(kuratowski=True, boundary=[1, 2, 3])
(True, None)
sage: g439.get_embedding()
{1: [7, 5],
2: [5, 6],
3: [6, 7],
4: [7, 6, 5],
5: [1, 4, 2],
6: [2, 4, 3],
7: [3, 4, 1]}
```

Order matters:

```
sage: K23 = graphs.CompleteBipartiteGraph(2, 3)
sage: K23.is_circular_planar(boundary=[0, 1, 2, 3])
True
sage: K23.is_circular_planar(ordered=True, boundary=[0, 1, 2, 3])
False
```

With a different order:

```
sage: K23.is_circular_planar(set_embedding=True, boundary=[0, 2, 1, 3])
True
```

is_clique(*vertices=None*, *directed_clique=False*, *induced=True*, *loops=False*)

Check whether a set of vertices is a clique

A clique is a set of vertices such that there is exactly one edge between any two vertices.

INPUT:

- `vertices` – a single vertex or an iterable container of vertices (default: `None`); when set, check whether the set of vertices is a clique, otherwise check whether ``self`` is a clique
- `directed_clique` – boolean (default: `False`); if set to `False`, only consider the underlying undirected graph. If set to `True` and the graph is directed, only return `True` if all possible edges in _both_ directions exist.
- `induced` – boolean (default: `True`); if set to `True`, check that the graph has exactly one edge between any two vertices. If set to `False`, check that the graph has at least one edge between any two vertices.
- `loops` – boolean (default: `False`); if set to `True`, check that each vertex of the graph has a loop, and exactly one if furthermore `induced == True`. If set to `False`, check that the graph has no loop when `induced == True`, and ignore loops otherwise.

EXAMPLES:

```
sage: g = graphs.CompleteGraph(4)
sage: g.is_clique([1, 2, 3])
True
sage: g.is_clique()
True
sage: h = graphs.CycleGraph(4)
```

(continues on next page)

(continued from previous page)

```
sage: h.is_clique([1, 2])
True
sage: h.is_clique([1, 2, 3])
False
sage: h.is_clique()
False
sage: i = digraphs.Complete(4)
sage: i.delete_edge([0, 1])
sage: i.is_clique(directed_clique=False, induced=True)
False
sage: i.is_clique(directed_clique=False, induced=False)
True
sage: i.is_clique(directed_clique=True)
False
```

is_connected(*G*)

Check whether the (di)graph is connected.

Note that in a graph, path connected is equivalent to connected.

INPUT:

- `G` – the input graph

See also:

- *is_biconnected()*

EXAMPLES:

```
sage: from sage.graphs.connectivity import is_connected
sage: G = Graph({0: [1, 2], 1: [2], 3: [4, 5], 4: [5]})
sage: is_connected(G)
False
sage: G.is_connected()
False
sage: G.add_edge(0,3)
sage: is_connected(G)
True
sage: D = DiGraph({0: [1, 2], 1: [2], 3: [4, 5], 4: [5]})
sage: is_connected(D)
False
sage: D.add_edge(0, 3)
sage: is_connected(D)
True
sage: D = DiGraph({1: [0], 2: [0]})
sage: is_connected(D)
True
```

is_cut_edge(*G*, *u*, *v=None*, *label=None*)

Returns True if the input edge is a cut-edge or a bridge.

A cut edge (or bridge) is an edge that when removed increases the number of connected components. This function works with simple graphs as well as graphs with loops and multiedges. In a digraph, a cut edge is an edge that when removed increases the number of (weakly) connected components.

INPUT: The following forms are accepted

- is_cut_edge(G, 1, 2)

- is_cut_edge(G, (1, 2))
- is_cut_edge(G, 1, 2, 'label')
- is_cut_edge(G, (1, 2, 'label'))

OUTPUT:

- Returns True if (u,v) is a cut edge, False otherwise

EXAMPLES:

```
sage: from sage.graphs.connectivity import is_cut_edge
sage: G = graphs.CompleteGraph(4)
sage: is_cut_edge(G,0,2)
False
sage: G.is_cut_edge(0,2)
False

sage: G = graphs.CompleteGraph(4)
sage: G.add_edge((0,5,'silly'))
sage: is_cut_edge(G,(0,5,'silly'))
True

sage: G = Graph([[0,1],[0,2],[3,4],[4,5],[3,5]])
sage: is_cut_edge(G,(0,1))
True

sage: G = Graph([[0,1],[0,2],[1,1]], loops = True)
sage: is_cut_edge(G,(1,1))
False

sage: G = digraphs.Circuit(5)
sage: is_cut_edge(G,(0,1))
False

sage: G = graphs.CompleteGraph(6)
sage: is_cut_edge(G,(0,7))
Traceback (most recent call last):
...
ValueError: edge not in graph
```

is_cut_vertex(*G*, *u*, *weak=False*)

Check whether the input vertex is a cut-vertex.

A vertex is a cut-vertex if its removal from the (di)graph increases the number of (strongly) connected components. Isolated vertices or leafs are not cut-vertices. This function works with simple graphs as well as graphs with loops and multiple edges.

INPUT:

- `G` – a Sage (Di)Graph
- `u` – a vertex
- `weak` – boolean (default: `False`); whether the connectivity of directed graphs is to be taken in the weak sense, that is ignoring edges orientations

OUTPUT:

Return `True` if `u` is a cut-vertex, and `False` otherwise.

EXAMPLES:

Giving a LollipopGraph(4,2), that is a complete graph with 4 vertices with a pending edge:

```
sage: from sage.graphs.connectivity import is_cut_vertex
sage: G = graphs.LollipopGraph(4, 2)
sage: is_cut_vertex(G, 0)
False
sage: is_cut_vertex(G, 3)
True
sage: G.is_cut_vertex(3)
True
```

Comparing the weak and strong connectivity of a digraph:

```
sage: from sage.graphs.connectivity import is_strongly_connected
sage: D = digraphs.Circuit(6)
sage: is_strongly_connected(D)
True
sage: is_cut_vertex(D, 2)
True
sage: is_cut_vertex(D, 2, weak=True)
False
```

Giving a vertex that is not in the graph:

```
sage: G = graphs.CompleteGraph(4)
sage: is_cut_vertex(G, 7)
Traceback (most recent call last):
...
ValueError: vertex (7) is not a vertex of the graph
```

is_cycle(*directed_cycle=True*)

Check whether `self` is a (directed) cycle graph.

We follow the definition provided in [BM2008] for undirected graphs. A cycle on three or more vertices is a simple graph whose vertices can be arranged in a cyclic order so that two vertices are adjacent if they are consecutive in the order, and not adjacent otherwise. A cycle on a vertex consists of a single vertex provided with a loop and a cycle with two vertices consists of two vertices connected by a pair of parallel edges. In other words, an undirected graph is a cycle if it is 2-regular and connected. The empty graph is not a cycle.

For directed graphs, a directed cycle, or circuit, on two or more vertices is a strongly connected directed graph without loops nor multiple edges with has many arcs as vertices. A circuit on a vertex consists of a single vertex provided with a loop.

INPUT:

- `directed_cycle` – boolean (default `True`); if set to `True` and the graph is directed, only return `True` if `self` is a directed cycle graph (i.e., a circuit). If set to `False`, we ignore the direction of edges and so opposite arcs become multiple (parallel) edges. This parameter is ignored for undirected graphs.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.is_cycle()
False
sage: graphs.CycleGraph(5).is_cycle()
True
sage: Graph([(0,1 )]).is_cycle()
```

(continues on next page)

(continued from previous page)

```
False
sage: Graph([(0, 1), (0, 1)], multiedges=True).is_cycle()
True
sage: Graph([(0, 1), (0, 1), (0, 1)], multiedges=True).is_cycle()
False
sage: Graph().is_cycle()
False
sage: G = Graph([(0, 0)], loops=True)
sage: G.is_cycle()
True
sage: digraphs.Circuit(3).is_cycle()
True
sage: digraphs.Circuit(2).is_cycle()
True
sage: digraphs.Circuit(2).is_cycle(directed_cycle=False)
True
sage: D = DiGraph(graphs.CycleGraph(3))
sage: D.is_cycle()
False
sage: D.is_cycle(directed_cycle=False)
False
sage: D.edges(labels=False)
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

is_drawn_free_of_edge_crossings()

Check whether the position dictionary for this graph is set and that position dictionary gives a planar embedding.

This simply checks all pairs of edges that don't share a vertex to make sure that they don't intersect.

Note: This function require that _pos attribute is set (Returns False otherwise)

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: pos = D.layout(layout='planar', save_pos=True)
sage: D.is_drawn_free_of_edge_crossings()
True
```

is_equitable(*partition*, *quotient_matrix=False*)

Checks whether the given partition is equitable with respect to self.

A partition is equitable with respect to a graph if for every pair of cells C1, C2 of the partition, the number of edges from a vertex of C1 to C2 is the same, over all vertices in C1.

INPUT:

- partition - a list of lists
- quotient_matrix - (default False) if True, and the partition is equitable, returns a matrix over the integers whose rows and columns represent cells of the partition, and whose i,j entry is the number of vertices in cell j adjacent to each vertex in cell i (since the partition is equitable, this is well defined)

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8],[7]])
```

(continues on next page)

(continued from previous page)

```
False
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8,7]])
True
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8,7]], quotient_matrix=True)
[1 2 0]
[1 0 2]
[0 2 1]
```

```
sage: ss = (graphs.WheelGraph(6)).line_graph(labels=False)
sage: prt = [[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3,␣
↪4)]]
```

```
sage: ss.is_equitable(prt)
Traceback (most recent call last):
...
TypeError: Partition ([[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)],␣
↪[(2, 3), (3, 4)]]) is not valid for this graph: vertices are incorrect.
```

```
sage: ss = (graphs.WheelGraph(5)).line_graph(labels=False)
sage: ss.is_equitable(prt)
False
```

is_eulerian(*path=False*)

Check whether the graph is Eulerian.

A graph is Eulerian if it has a (closed) tour that visits each edge exactly once.

INPUT:

- `path` – boolean (default: `False`); by default this function finds if the graph contains a closed tour visiting each edge once, i.e. an Eulerian cycle. If you want to test the existence of an Eulerian path, set this argument to `True`. Graphs with this property are sometimes called semi-Eulerian.

OUTPUT:

`True` or `False` for the closed tour case. For an open tour search (`path``=``True`) the function returns `False` if the graph is not semi-Eulerian, or a tuple (u, v) in the other case. This tuple defines the edge that would make the graph Eulerian, i.e. close an existing open tour. This edge may or may not be already present in the graph.

EXAMPLES:

```
sage: graphs.CompleteGraph(4).is_eulerian()
False
sage: graphs.CycleGraph(4).is_eulerian()
True
sage: g = DiGraph({0:[1,2], 1:[2]}); g.is_eulerian()
False
sage: g = DiGraph({0:[2], 1:[3], 2:[0,1], 3:[2]}); g.is_eulerian()
True
sage: g = DiGraph({0:[1], 1:[2], 2:[0], 3:[]}); g.is_eulerian()
True
sage: g = Graph([(1,2), (2,3), (3,1), (4,5), (5,6), (6,4)]); g.is_eulerian()
False
```

```
sage: g = DiGraph({0: [1]}); g.is_eulerian(path=True)
(1, 0)
```

(continues on next page)

(continued from previous page)

```
sage: graphs.CycleGraph(4).is_eulerian(path=True)
False
sage: g = DiGraph({0: [1], 1: [2,3], 2: [4]}); g.is_eulerian(path=True)
False
```

```
sage: g = Graph({0:[1,2,3], 1:[2,3], 2:[3,4], 3:[4]}, multiedges=True)
sage: g.is_eulerian()
False
sage: e = g.is_eulerian(path=True); e
(0, 1)
sage: g.add_edge(e)
sage: g.is_eulerian(path=False)
True
sage: g.is_eulerian(path=True)
False
```

is_gallai_tree()

Return whether the current graph is a Gallai tree.

A graph is a Gallai tree if and only if it is connected and its 2-connected components are all isomorphic to complete graphs or odd cycles.

A connected graph is not degree-choosable if and only if it is a Gallai tree [ERT1979].

EXAMPLES:

A complete graph is, or course, a Gallai Tree:

```
sage: g = graphs.CompleteGraph(15)
sage: g.is_gallai_tree()
True
```

The Petersen Graph is not:

```
sage: g = graphs.PetersenGraph()
sage: g.is_gallai_tree()
False
```

A Graph built from vertex-disjoint complete graphs linked by one edge to a special vertex -1 is a ''star-shaped'' Gallai tree:

```
sage: g = 8 * graphs.CompleteGraph(6)
sage: g.add_edges([(-1, c[0]) for c in g.connected_components()])
sage: g.is_gallai_tree()
True
```

is_hamiltonian(*solver=None*, *constraint_generation=None*, *verbose=0*, *verbose_constraints=False*)

Test whether the current graph is Hamiltonian.

A graph (resp. digraph) is said to be Hamiltonian if it contains as a subgraph a cycle (resp. a circuit) going through all the vertices.

Testing for Hamiltonicity being NP-Complete, this algorithm could run for some time depending on the instance.

ALGORITHM:

See *traveling_salesman_problem()*.

INPUT:

- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `constraint_generation` (boolean) – whether to use constraint generation when solving the Mixed Integer Linear Program. When `constraint_generation = None`, constraint generation is used whenever the graph has a density larger than 70%.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.
- `verbose_constraints` – whether to display which constraints are being generated.

OUTPUT:

Returns `True` if a Hamiltonian cycle/circuit exists, and `False` otherwise.

NOTE:

This function, as `hamiltonian_cycle` and `traveling_salesman_problem`, computes a Hamiltonian cycle if it exists: the user should *NOT* test for Hamiltonicity using `is_hamiltonian` before calling `hamiltonian_cycle` or `traveling_salesman_problem` as it would result in computing it twice.

EXAMPLES:

The Heawood Graph is known to be Hamiltonian

```
sage: g = graphs.HeawoodGraph()
sage: g.is_hamiltonian()
True
```

The Petergraph, though, is not

```
sage: g = graphs.PetersenGraph()
sage: g.is_hamiltonian()
False
```

is_immutable()

Check whether the graph is immutable.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.is_immutable()
False
sage: Graph(G, immutable=True).is_immutable()
True
```

is_independent_set(*vertices=None*)

Check whether `vertices` is an independent set of `self`.

An independent set is a set of vertices such that there is no edge between any two vertices.

INPUT:

- `vertices` – a single vertex or an iterable container of vertices (default: `None`); when set, check whether the given set of vertices is an independent set, otherwise, check whether the set of vertices of ``self is an independent set

EXAMPLES:

```
sage: graphs.CycleGraph(4).is_independent_set([1,3])
True
sage: graphs.CycleGraph(4).is_independent_set([1,2,3])
False
```

is_interval(*certificate=False*)

Check whether the graph is an interval graph.

An *interval graph* is one where every vertex can be seen as an interval on the real line so that there is an edge in the graph iff the corresponding intervals intersects.

See the Wikipedia article Interval_graph for more information.

INPUT:

- `certificate` – boolean (default: `False`);
  - When `certificate=False`, returns `True` is the graph is an interval graph and `False` otherwise
  - When `certificate=True`, returns either `(False, None)` or `(True, d)` where `d` is a dictionary whose keys are the vertices and values are pairs of integers. They correspond to an embedding of the interval graph, each vertex being represented by an interval going from the first of the two values to the second.

ALGORITHM:

Through the use of PQ-Trees.

AUTHOR:

Nathann Cohen (implementation)

EXAMPLES:

```
sage: g = Graph({1: [2, 3, 4], 4: [2, 3]})
sage: g.is_interval()
True
sage: g.is_interval(certificate=True)
(True, {1: (0, 5), 2: (4, 6), 3: (1, 3), 4: (2, 7)})
```

The Petersen Graph is not chordal, so it cannot be an interval graph:

```
sage: g = graphs.PetersenGraph()
sage: g.is_interval()
False
```

A chordal but still not an interval graph:

```
sage: g = Graph({1: [4, 2, 3], 2: [3, 5], 3: [6]})
sage: g.is_interval()
False
```

See also:

- *Interval Graph Recognition*.
- *PQ* – implementation of PQ-Trees
- *is_chordal()*
- *IntervalGraph()*

- *RandomIntervalGraph()*

is_isomorphic(*other*, *certificate=False*, *verbosity=0*, *edge_labels=False*)
Tests for isomorphism between self and other.

INPUT:

- `certificate` - if True, then output is (a, b), where a is a boolean and b is either a map or `None`.
- `edge_labels` - default `False`, otherwise allows only permutations respecting edge labels.

OUTPUT:

- either a boolean or, if `certificate` is `True`, a tuple consisting of a boolean and a map or `None`

EXAMPLES:

Graphs:

```
sage: from sage.groups.perm_gps.permgroup_named import SymmetricGroup
sage: D = graphs.DodecahedralGraph()
sage: E = copy(D)
sage: gamma = SymmetricGroup(20).random_element()
sage: E.relabel(gamma)
sage: D.is_isomorphic(E)
True
```

```
sage: D = graphs.DodecahedralGraph()
sage: S = SymmetricGroup(20)
sage: gamma = S.random_element()
sage: E = copy(D)
sage: E.relabel(gamma)
sage: a,b = D.is_isomorphic(E, certificate=True); a
True
sage: from sage.graphs.generic_graph_pyx import spring_layout_fast
sage: position_D = spring_layout_fast(D)
sage: position_E = {}
sage: for vert in position_D:
....:  position_E[b[vert]] = position_D[vert]
sage: graphics_array([D.plot(pos=position_D), E.plot(pos=position_E)]).show()
↪# long time
```

```
sage: g=graphs.HeawoodGraph()
sage: g.is_isomorphic(g)
True
```

Multigraphs:

```
sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edge((0,1,1))
sage: G.add_edge((0,1,2))
sage: G.add_edge((0,1,3))
sage: G.add_edge((0,1,4))
sage: H = Graph(multiedges=True,sparse=True)
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: G.is_isomorphic(H)
True
```

Digraphs:

```
sage: A = DiGraph( { 0 : [1,2] } )
sage: B = DiGraph( { 1 : [0,2] } )
sage: A.is_isomorphic(B, certificate=True)
(True, {0: 1, 1: 0, 2: 2})
```

Edge labeled graphs:

```
sage: G = Graph(sparse=True)
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: H = G.relabel([1,2,3,4,0], inplace=False)
sage: G.is_isomorphic(H, edge_labels=True)
True
```

Edge labeled digraphs:

```
sage: G = DiGraph()
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: H = G.relabel([1,2,3,4,0], inplace=False)
sage: G.is_isomorphic(H, edge_labels=True)
True
sage: G.is_isomorphic(H, edge_labels=True, certificate=True)
(True, {0: 1, 1: 2, 2: 3, 3: 4, 4: 0})
```

is_planar(*on_embedding=None*, *kuratowski=False*, *set_embedding=False*, *set_pos=False*)

Check whether the graph is planar.

This wraps the reference implementation provided by John Boyer of the linear time planarity algorithm by edge addition due to Boyer Myrvold. (See reference code in *planarity*).

Note: The argument on_embedding takes precedence over `set_embedding`. This means that only the `on_embedding` combinatorial embedding will be tested for planarity and no `_embedding` attribute will be set as a result of this function call, unless `on_embedding` is None.

REFERENCE:

[BM2004]

See also:

- "Almost planar graph": *is_apex()*
- "Measuring non-planarity": *genus()*, *crossing_number()*
- *planar_dual()*
- *faces()*
- *is_polyhedral()*

INPUT:

- `on_embedding` – dictionary (default: `None`); the embedding dictionary to test planarity on (i.e.: will return `True` or `False` only for the given embedding)
- `kuratowski` – boolean (default: `False`); whether to return a tuple with boolean as first entry. If the graph is nonplanar, will return the Kuratowski subgraph (i.e. an edge subdivision of K_5 or $K_{3,3}$) as

the second tuple entry. If the graph is planar, returns `None` as the second entry. When set to `False`, only a boolean answer is returned.

- `set_embedding` – boolean (default: `False`); whether to set the instance field variable that contains a combinatorial embedding (clockwise ordering of neighbors at each vertex). This value will only be set if a planar embedding is found. It is stored as a Python dict: `v1 : [n1,n2,n3]` where `v1` is a vertex and `n1,n2,n3` are its neighbors.
- `set_pos` – boolean (default: `False`); whether to set the position dictionary (for plotting) to reflect the combinatorial embedding. Note that this value will default to False if set_emb is set to False. Also, the position dictionary will only be updated if a planar embedding is found.

EXAMPLES:

```
sage: g = graphs.CubeGraph(4)
sage: g.is_planar()
False
```

```
sage: g = graphs.CircularLadderGraph(4)
sage: g.is_planar(set_embedding=True)
True
sage: g.get_embedding()
{0: [1, 4, 3],
 1: [2, 5, 0],
 2: [3, 6, 1],
 3: [0, 7, 2],
 4: [0, 5, 7],
 5: [1, 6, 4],
 6: [2, 7, 5],
 7: [4, 6, 3]}
```

```
sage: g = graphs.PetersenGraph()
sage: (g.is_planar(kuratowski=True))[1].adjacency_matrix()
[0 1 0 0 0 1 0 0 0]
[1 0 1 0 0 0 1 0 0]
[0 1 0 1 0 0 0 1 0]
[0 0 1 0 0 0 0 0 1]
[0 0 0 0 0 0 1 1 0]
[1 0 0 0 0 0 0 1 1]
[0 1 0 0 1 0 0 0 1]
[0 0 1 0 1 1 0 0 0]
[0 0 0 1 0 1 1 0 0]
```

```
sage: k43 = graphs.CompleteBipartiteGraph(4, 3)
sage: result = k43.is_planar(kuratowski=True); result
(False, Graph on 6 vertices)
sage: result[1].is_isomorphic(graphs.CompleteBipartiteGraph(3, 3))
True
```

Multi-edged and looped graphs are partially supported:

```
sage: G = Graph({0: [1, 1]}, multiedges=True)
sage: G.is_planar()
True
sage: G.is_planar(on_embedding={})
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
NotImplementedError: cannot compute with embeddings of multiple-edged or␣
↪looped graphs
sage: G.is_planar(set_pos=True)
Traceback (most recent call last):
...
NotImplementedError: cannot compute with embeddings of multiple-edged or␣
↪looped graphs
sage: G.is_planar(set_embedding=True)
Traceback (most recent call last):
...
NotImplementedError: cannot compute with embeddings of multiple-edged or␣
↪looped graphs
sage: G.is_planar(kuratowski=True)
(True, None)
```

```
sage: G = graphs.CompleteGraph(5)
sage: G = Graph(G, multiedges=True)
sage: G.add_edge(0, 1)
sage: G.is_planar()
False
sage: b,k = G.is_planar(kuratowski=True)
sage: b
False
sage: k.vertices()
[0, 1, 2, 3, 4]
```

is_regular(*k=None*)

Check whether this graph is (k-)regular.

INPUT:

- `k` – integer (default: `None`); the degree of regularity to check for

EXAMPLES:

```
sage: G = graphs.HoffmanSingletonGraph()
sage: G.is_regular()
True
sage: G.is_regular(9)
False
```

So the Hoffman-Singleton graph is regular, but not 9-regular. In fact, we can now find the degree easily as follows:

```
sage: next(G.degree_iterator())
7
```

The house graph is not regular:

```
sage: graphs.HouseGraph().is_regular()
False
```

A graph without vertices is k-regular for every k:

```
sage: Graph().is_regular()
True
```

is_self_complementary()
Check whether the graph is self-complementary.

A (di)graph is self-complementary if it is isomorphic to its (di)graph complement. For instance, the path graph P_4 and the cycle graph C_5 are self-complementary.

See also:

- Wikipedia article Self-complementary_graph
- OEIS sequence A000171 for the numbers of self-complementary graphs of order n
- OEIS sequence A003086 for the numbers of self-complementary digraphs of order n.

EXAMPLES:

The only self-complementary path graph is P_4:

```
sage: graphs.PathGraph(4).is_self_complementary()
True
sage: graphs.PathGraph(5).is_self_complementary()
False
```

The only self-complementary directed path is P_2:

```
sage: digraphs.Path(2).is_self_complementary()
True
sage: digraphs.Path(3).is_self_complementary()
False
```

Every Paley graph is self-complementary:

```
sage: G = graphs.PaleyGraph(9)
sage: G.is_self_complementary()
True
```

is_subgraph(*other*, *induced=True*)
Check whether `self` is a subgraph of `other`.

Warning: Please note that this method does not check whether `self` contains a subgraph *isomorphic* to `other`, but only if it directly contains it as a subgraph !

By default `induced` is `True` for backwards compatibility.

INPUT:

- `other` – a Sage (Di)Graph
- `induced` - boolean (default: `True`); if set to `True` check whether the graph is an *induced* subgraph of `other` that is if the vertices of the graph are also vertices of `other`, and the edges of the graph are equal to the edges of `other` between the vertices contained in the graph.

  If set to `False` tests whether the graph is a subgraph of `other` that is if all vertices of the graph are also in `other` and all edges of the graph are also in `other`.

OUTPUT:

boolean – `True` iff the graph is a (possibly induced) subgraph of `other`.

See also:

If you are interested in the (possibly induced) subgraphs isomorphic to the graph in `other`, you are looking for the following methods:

- *subgraph_search()* – find a subgraph isomorphic to `other` inside of the graph
- *subgraph_search_count()* – count the number of such copies
- *subgraph_search_iterator()* – iterator over all the copies of `other` contained in the graph

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: G = P.subgraph(range(6))
sage: G.is_subgraph(P)
True

sage: H = graphs.CycleGraph(5)
sage: G = graphs.PathGraph(5)
sage: G.is_subgraph(H)
False
sage: G.is_subgraph(H, induced=False)
True
sage: H.is_subgraph(G, induced=False)
False
```

is_transitively_reduced()

Check whether the digraph is transitively reduced.

A digraph is transitively reduced if it is equal to its transitive reduction. A graph is transitively reduced if it is a forest.

EXAMPLES:

```
sage: d = DiGraph({0: [1], 1: [2], 2: [3]})
sage: d.is_transitively_reduced()
True

sage: d = DiGraph({0: [1, 2], 1: [2]})
sage: d.is_transitively_reduced()
False

sage: d = DiGraph({0: [1, 2], 1: [2], 2: []})
sage: d.is_transitively_reduced()
False
```

is_vertex_transitive(*partition=None*, *verbosity=0*, *edge_labels=False*, *order=False*, *return_group=True*, *orbits=False*)

Returns whether the automorphism group of self is transitive within the partition provided, by default the unit partition of the vertices of self (thus by default tests for vertex transitivity in the usual sense).

EXAMPLES:

```
sage: G = Graph({0:[1],1:[2]})
sage: G.is_vertex_transitive()
False
sage: P = graphs.PetersenGraph()
sage: P.is_vertex_transitive()
True
sage: D = graphs.DodecahedralGraph()
sage: D.is_vertex_transitive()
```

(continues on next page)

(continued from previous page)

```
True
sage: R = graphs.RandomGNP(2000, .01)
sage: R.is_vertex_transitive()
False
```

katz_centrality(*alpha*, *u=None*)

Return the Katz centrality of vertex u.

Katz centrality of a node is a measure of centrality in a graph network. Katz centrality computes the relative influence of a node within a network. Connections made with distant neighbors are, however penalized by an attenuation factor α.

See the Wikipedia article Katz_centrality for more information.

INPUT:

- `alpha` – a nonnegative real number, must be less than the reciprocal of the spectral radius of the graph (the maximum absolute eigenvalue of the adjacency matrix).
- `u` – the vertex whose Katz centrality needs to be measured (default: `None`)

OUTPUT: a list containing the Katz centrality of each vertex if u=None otherwise Katz centrality of the vertex u.

EXAMPLES:

We compute the Katz centrality of a 4-cycle (note that by symmetry, all 4 vertices have the same centrality)

```
sage: G = graphs.CycleGraph(4)
sage: G.katz_centrality(1/20)
{0: 1/9, 1: 1/9, 2: 1/9, 3: 1/9}
```

Note that in the below example the nodes having indegree 0 also have the Katz centrality value as 0, as these nodes are not influenced by other nodes.

```
sage: G = DiGraph({1: [10], 2:[10,11], 3:[10,11], 4:[], 5:[11, 4], 6:[11],␣
↪7:[10,11], 8:[10,11], 9:[10], 10:[11, 5, 8], 11:[6]})
sage: G.katz_centrality(.85)
{1: 0.000000000000000,
 2: 0.000000000000000,
 3: 0.000000000000000,
 4: 16.7319819819820,
 5: 18.6846846846847,
 6: 173.212076941807,
 7: 0.000000000000000,
 8: 18.6846846846847,
 9: 0.000000000000000,
 10: 20.9819819819820,
 11: 202.778914049184}
```

See also:

- *katz_matrix()*
- Wikipedia article Katz_centrality

katz_matrix(*alpha*, *nonedgesonly=False*, *vertices=None*)

Return the Katz matrix of the graph.

Katz centrality of a node is a measure of centrality in a graph network. Katz centrality computes the relative influence of a node within a network. Connections made with distant neighbors are, however penalized by an attenuation factor α.

Adding the values in the Katz matrix of all columns in a particular row gives the Katz centrality measure of the vertex represented by that particular row. Katz centrality measures influence by taking into account the total number of walks between a pair of nodes.

See the Wikipedia article Katz_centrality for more information.

INPUT:

- `alpha` – a nonnegative real number, must be less than the reciprocal of the spectral radius of the graph (the maximum absolute eigenvalue of the adjacency matrix)
- `nonedgesonly` – boolean (default: `True`); if `True`, value for each edge present in the graph is set to zero.
- `vertices` – list (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *`GenericGraph.vertices()`* is used.

OUTPUT: the Katz matrix of the graph with parameter alpha

EXAMPLES:

We find the Katz matrix of an undirected 4-cycle.

```
sage: G = graphs.CycleGraph(4)
sage: G.katz_matrix(1/20)
[1/198  5/99 1/198  5/99]
[ 5/99 1/198  5/99 1/198]
[1/198  5/99 1/198  5/99]
[ 5/99 1/198  5/99 1/198]
```

We find the Katz matrix of an undirected 4-cycle with all entries other than those which correspond to non-edges zeroed out.

```
sage: G.katz_matrix(1/20, True)
[    0     0 1/198     0]
[    0     0     0 1/198]
[1/198     0     0     0]
[    0 1/198 0     0]
```

This will give an error if alpha<=0 or alpha>=1/spectral_radius = 1/max (A.eigenvalues()).

We find the Katz matrix in a fan on 6 vertices.

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.katz_matrix(1/10)
[   169/2256    545/4512      25/188    605/4512      25/188    545/4512
↪485/4512]
[   545/4512 7081/297792  4355/37224    229/9024   595/37224 4073/297792
↪109/9024]
[     25/188  4355/37224    172/4653      45/376    125/4653   595/37224
↪ 5/376]
[   605/4512    229/9024      45/376    337/9024      45/376    229/9024
↪121/9024]
[     25/188   595/37224    125/4653      45/376    172/4653  4355/37224
↪ 5/376]
[   545/4512 4073/297792   595/37224    229/9024  4355/37224 7081/297792
↪109/9024]
```

(continues on next page)

(continued from previous page)

```
[   485/4512    109/9024       5/376    121/9024       5/376    109/9024
↪97/9024]
```

See also:

- *katz_centrality()*
- Wikipedia article Katz_centrality

kirchhoff_matrix(*weighted=None*, *indegree=True*, *normalized=False*, *signless=False*, ***kwds*)

Return the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be $D + M$ if signless and $D - M$ otherwise, where D is the diagonal degree matrix (each diagonal entry is the degree of the corresponding vertex), and M is the adjacency matrix. If `normalized` is `True`, then the returned matrix is $D^{-1/2}(D + M)D^{-1/2}$ if signless and $D^{-1/2}(D - M)D^{-1/2}$ otherwise.

(In the special case of DiGraphs, D is defined as the diagonal in-degree matrix or diagonal out-degree matrix according to the value of `indegree`)

INPUT:

- `weighted` – boolean (default: `None`);
  - If `True`, the weighted adjacency matrix is used for M, and the diagonal matrix D takes into account the weight of edges (replace in the definition "degree" by "sum of the incident edges")
  - Else, each edge is assumed to have weight 1

  Default is to take weights into consideration if and only if the graph is weighted.
- `indegree` – boolean (default: `True`); this parameter is considered only for digraphs.
  - If `True`, each diagonal entry of D is equal to the in-degree of the corresponding vertex
  - Else, each diagonal entry of D is equal to the out-degree of the corresponding vertex

  By default, `indegree` is set to `True`
- `normalized` – boolean (default: `False`);
  - If `True`, the returned matrix is $D^{-1/2}(D + M)D^{-1/2}$ for signless and $D^{-1/2}(D - M)D^{-1/2}$ otherwise, a normalized version of the Laplacian matrix. More accurately, the normalizing matrix used is equal to $D^{-1/2}$ only for non-isolated vertices. If vertex i is isolated, then diagonal entry i in the matrix is 1, rather than a division by zero
  - Else, the matrix $D + M$ for signless and $D - M$ otherwise is returned
- `signless` – boolean (default: `False`);
  - If `True`, $D + M$ is used in calculation of Kirchhoff matrix
  - Else, $D - M$ is used in calculation of Kirchhoff matrix

Note that any additional keywords will be passed on to either the `adjacency_matrix` or `weighted_adjacency_matrix` method.

AUTHORS:

- Tom Boothby
- Jason Grout

EXAMPLES:

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0, 1, 1), (1, 2, 2), (0, 2, 3), (0, 3, 4)])
sage: M = G.kirchhoff_matrix(weighted=True); M
[ 8 -1 -3 -4]
[-1  3 -2  0]
[-3 -2  5  0]
[-4  0  0  4]
sage: M = G.kirchhoff_matrix(); M
[ 3 -1 -1 -1]
[-1  2 -1  0]
[-1 -1  2  0]
[-1  0  0  1]
sage: M = G.laplacian_matrix(normalized=True); M
[                   1 -1/6*sqrt(3)*sqrt(2) -1/6*sqrt(3)*sqrt(2)          -1/
↪3*sqrt(3)]
[-1/6*sqrt(3)*sqrt(2)                    1                 -1/2
↪      0]
[-1/6*sqrt(3)*sqrt(2)                 -1/2                    1
↪      0]
[         -1/3*sqrt(3)                    0                    0
↪      1]
sage: M = G.kirchhoff_matrix(weighted=True, signless=True); M
[8 1 3 4]
[1 3 2 0]
[3 2 5 0]
[4 0 0 4]

sage: G = Graph({0: [], 1: [2]})
sage: G.laplacian_matrix(normalized=True)
[ 0  0  0]
[ 0  1 -1]
[ 0 -1  1]
sage: G.laplacian_matrix(normalized=True,signless=True)
[0 0 0]
[0 1 1]
[0 1 1]
```

A weighted directed graph with loops, changing the variable `indegree`

```
sage: G = DiGraph({1: {1: 2, 2: 3}, 2: {1: 4}}, weighted=True, sparse=True)
sage: G.laplacian_matrix()
[ 4 -3]
[-4  3]
```

```
sage: G = DiGraph({1: {1: 2, 2: 3}, 2: {1: 4}}, weighted=True, sparse=True)
sage: G.laplacian_matrix(indegree=False)
[ 3 -3]
[-4  4]
```

A different ordering of the vertices (see *adjacency_matrix()* and *weighted_adjacency_matrix()*):

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0, 1, 1), (1, 2, 2), (0, 2, 3), (0, 3, 4)])
sage: M = G.kirchhoff_matrix(vertices=[3, 2, 1, 0]); M
[ 1  0  0 -1]
[ 0  2 -1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 0 -1  2 -1]
[-1 -1 -1  3]
sage: M = G.kirchhoff_matrix(weighted=True, vertices=[3, 2, 1, 0]); M
[ 4  0  0 -4]
[ 0  5 -2 -3]
[ 0 -2  3 -1]
[-4 -3 -1  8]
```

kronecker_product(*other*)

Return the tensor product of `self` and `other`.

The tensor product of G and H is the graph L with vertex set $V(L)$ equal to the Cartesian product of the vertices $V(G)$ and $V(H)$, and $((u, v), (w, x))$ is an edge iff - (u, w) is an edge of self, and - (v, x) is an edge of other.

The tensor product is also known as the categorical product and the kronecker product (refering to the kronecker matrix product). See the Wikipedia article Kronecker_product.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: T = C.tensor_product(Z); T
Graph on 10 vertices
sage: T.size()
10
sage: T.plot() # long time
Graphics object consisting of 21 graphics primitives
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: T = D.tensor_product(P); T
Graph on 200 vertices
sage: T.size()
900
sage: T.plot() # long time
Graphics object consisting of 1101 graphics primitives
```

laplacian_matrix(*weighted=None*, *indegree=True*, *normalized=False*, *signless=False*, ***kwds*)

Return the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be $D + M$ if signless and $D - M$ otherwise, where D is the diagonal degree matrix (each diagonal entry is the degree of the corresponding vertex), and M is the adjacency matrix. If `normalized` is `True`, then the returned matrix is $D^{-1/2}(D + M)D^{-1/2}$ if signless and $D^{-1/2}(D - M)D^{-1/2}$ otherwise.

(In the special case of DiGraphs, D is defined as the diagonal in-degree matrix or diagonal out-degree matrix according to the value of `indegree`)

INPUT:

- `weighted` – boolean (default: `None`);
  - If `True`, the weighted adjacency matrix is used for M, and the diagonal matrix D takes into account the weight of edges (replace in the definition "degree" by "sum of the incident edges")
  - Else, each edge is assumed to have weight 1

  Default is to take weights into consideration if and only if the graph is weighted.

- `indegree` – boolean (default: `True`); this parameter is considered only for digraphs.
  - If `True`, each diagonal entry of D is equal to the in-degree of the corresponding vertex
  - Else, each diagonal entry of D is equal to the out-degree of the corresponding vertex

  By default, `indegree` is set to `True`
- `normalized` – boolean (default: `False`);
  - If `True`, the returned matrix is $D^{-1/2}(D + M)D^{-1/2}$ for signless and $D^{-1/2}(D - M)D^{-1/2}$ otherwise, a normalized version of the Laplacian matrix. More accurately, the normalizing matrix used is equal to $D^{-1/2}$ only for non-isolated vertices. If vertex i is isolated, then diagonal entry i in the matrix is 1, rather than a division by zero
  - Else, the matrix $D + M$ for signless and $D - M$ otherwise is returned
- `signless` – boolean (default: `False`);
  - If `True`, $D + M$ is used in calculation of Kirchhoff matrix
  - Else, $D - M$ is used in calculation of Kirchhoff matrix

Note that any additional keywords will be passed on to either the `adjacency_matrix` or `weighted_adjacency_matrix` method.

AUTHORS:

- Tom Boothby
- Jason Grout

EXAMPLES:

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0, 1, 1), (1, 2, 2), (0, 2, 3), (0, 3, 4)])
sage: M = G.kirchhoff_matrix(weighted=True); M
[ 8 -1 -3 -4]
[-1  3 -2  0]
[-3 -2  5  0]
[-4  0  0  4]
sage: M = G.kirchhoff_matrix(); M
[ 3 -1 -1 -1]
[-1  2 -1  0]
[-1 -1  2  0]
[-1  0  0  1]
sage: M = G.laplacian_matrix(normalized=True); M
[                   1 -1/6*sqrt(3)*sqrt(2) -1/6*sqrt(3)*sqrt(2)          -1/
↪3*sqrt(3)]
[-1/6*sqrt(3)*sqrt(2)                    1                 -1/2
↪      0]
[-1/6*sqrt(3)*sqrt(2)                 -1/2                    1
↪      0]
[         -1/3*sqrt(3)                    0                    0
↪      1]
sage: M = G.kirchhoff_matrix(weighted=True, signless=True); M
[8 1 3 4]
[1 3 2 0]
[3 2 5 0]
[4 0 0 4]

sage: G = Graph({0: [], 1: [2]})
sage: G.laplacian_matrix(normalized=True)
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0]
[ 0  1 -1]
[ 0 -1  1]
sage: G.laplacian_matrix(normalized=True,signless=True)
[0 0 0]
[0 1 1]
[0 1 1]
```

A weighted directed graph with loops, changing the variable `indegree`

```
sage: G = DiGraph({1: {1: 2, 2: 3}, 2: {1: 4}}, weighted=True, sparse=True)
sage: G.laplacian_matrix()
[ 4 -3]
[-4  3]
```

```
sage: G = DiGraph({1: {1: 2, 2: 3}, 2: {1: 4}}, weighted=True, sparse=True)
sage: G.laplacian_matrix(indegree=False)
[ 3 -3]
[-4  4]
```

A different ordering of the vertices (see *adjacency_matrix()* and *weighted_adjacency_matrix()*):

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0, 1, 1), (1, 2, 2), (0, 2, 3), (0, 3, 4)])
sage: M = G.kirchhoff_matrix(vertices=[3, 2, 1, 0]); M
[ 1  0  0 -1]
[ 0  2 -1 -1]
[ 0 -1  2 -1]
[-1 -1 -1  3]
sage: M = G.kirchhoff_matrix(weighted=True, vertices=[3, 2, 1, 0]); M
[ 4  0  0 -4]
[ 0  5 -2 -3]
[ 0 -2  3 -1]
[-4 -3 -1  8]
```

latex_options()

Return an instance of *GraphLatex* for the graph.

Changes to this object will affect the LaTeX version of the graph. For a full explanation of how to use LaTeX to render graphs, see the introduction to the *graph_latex* module.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: opts
LaTeX options for Petersen graph: {}
sage: opts.set_option('tkz_style', 'Classic')
sage: opts
LaTeX options for Petersen graph: {'tkz_style': 'Classic'}
```

layout(*layout=None*, *pos=None*, *dim=2*, *save_pos=False*, ***options*)

Return a layout for the vertices of this graph.

INPUT:

- `layout` – string (default: `None`); specifies a layout algorithm among `"acyclic"`, `"acyclic_dummy"`, `"circular"`, `"ranked"`, `"graphviz"`, `"planar"`, `"spring"`, or `"tree"`
- `pos` – dictionary (default: `None`); a dictionary of positions
- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3
- `save_pos` – boolean (default: `False`); whether to save the positions
- `**options` – layout options (see below)

If `layout` is set, the specified algorithm is used to compute the positions.

Otherwise, if `pos` is specified, use the given positions.

Otherwise, try to fetch previously computed and saved positions.

Otherwise use the default layout (usually the spring layout).

If `save_pos = True`, the layout is saved for later use.

EXAMPLES:

```
sage: g = digraphs.ButterflyGraph(1)
sage: D2 = g.layout(); D2  # random
{('0', 0): [2.69..., 0.43...],
 ('0', 1): [1.35..., 0.86...],
 ('1', 0): [0.89..., -0.42...],
 ('1', 1): [2.26..., -0.87...]}

sage: g.layout(layout="acyclic_dummy", save_pos=True)
{('0', 0): [0.3..., 0],
 ('0', 1): [0.3..., 1],
 ('1', 0): [0.6..., 0],
 ('1', 1): [0.6..., 1]}

sage: D3 = g.layout(dim=3); D3  # random
{('0', 0): [0.68..., 0.50..., -0.24...],
 ('0', 1): [1.02..., -0.02..., 0.93...],
 ('1', 0): [2.06..., -0.49..., 0.23...],
 ('1', 1): [1.74..., 0.01..., -0.92...]}
```

Some safety tests:

```
sage: sorted(D2.keys()) == sorted(D3.keys()) == sorted(g)
True
sage: isinstance(D2, dict) and isinstance(D3, dict)
True
sage: [c in RDF for c in D2[('0', 0)]]
[True, True]
sage: [c in RDF for c in D3[('0', 0)]]
[True, True, True]
```

Here is the list of all the available layout options (`**options`):

```
sage: from sage.graphs.graph_plot import layout_options
sage: for key, value in sorted(layout_options.items()):
....:     print("option {} : {}".format(key, value))
option by_component : Whether to do the spring layout by connected component -
↪- a boolean.
```

(continues on next page)

(continued from previous page)

```
option dim : The dimension of the layout -- 2 or 3.
option heights : A dictionary mapping heights to the list of vertices at this␣
↪height.
option iterations : The number of times to execute the spring layout␣
↪algorithm.
option layout : A layout algorithm -- one of : "acyclic", "circular" (plots␣
↪the graph with vertices evenly distributed on a circle), "ranked", "graphviz
↪", "planar", "spring" (traditional spring layout, using the graph's current␣
↪positions as initial positions), or "tree" (the tree will be plotted in␣
↪levels, depending on minimum distance for the root).
option prog : Which graphviz layout program to use -- one of "circo", "dot",
↪"fdp", "neato", or "twopi".
option save_pos : Whether or not to save the computed position for the graph.
option spring : Use spring layout to finalize the current layout.
option tree_orientation : The direction of tree branches -- 'up', 'down',
↪'left' or 'right'.
option tree_root : A vertex designation for drawing trees. A vertex of the␣
↪tree to be used as the root for the ``layout='tree'`` option. If no root is␣
↪specified, then one is chosen close to the center of the tree. Ignored␣
↪unless ``layout='tree'``
```

Some of them only apply to certain layout algorithms. For details, see *layout_acyclic()*, *layout_planar()*, *layout_circular()*, *layout_spring()*, ...

Warning: unknown optional arguments are silently ignored

Warning: `graphviz` and `dot2tex` are currently required to obtain a nice `'acyclic'` layout. See *layout_graphviz()* for installation instructions.

A subclass may implement another layout algorithm `"blah"`, by implementing a method `.layout_blah`. It may override the default layout by overriding *layout_default()*, and similarly override the predefined layouts.

Todo: use this feature for all the predefined graphs classes (like for the Petersen graph, ...), rather than systematically building the layout at construction time.

---

layout_circular(*dim=2*, *center=(0, 0)*, *radius=1*, *shift=0*, *angle=0*, ***options*)

Return a circular layout for this graph

INPUT:

- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3
- `center` – tuple (default: `(0, 0)`); position of the center of the circle
- `radius` – (default: 1); the radius of the circle
- `shift` – (default: 0); rotation of the circle. A value of `shift=1` will replace in the drawing the i-th element of the list by the $(i-1)$-th. Non-integer values are admissible, and a value of α corresponds to a rotation of the circle by an angle of $\alpha 2\pi/n$ (where n is the number of vertices set on the circle).
- `angle` – (default: 0); rotate the embedding of all vertices. For instance, when `angle == 0`, the first vertex get position `(center[0] + radius, center[1])`. With a value of $\pi/2$, the first

vertex get position `(center[0], center[1] + radius)`.

- `**options` – other parameters not used here

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: G = graphs.CirculantGraph(7, [1, 3])
sage: G.layout_circular()
{0: (0.0, 1.0),
 1: (-0.78...,  0.62...),
 2: (-0.97..., -0.22...),
 3: (-0.43..., -0.90...),
 4: (0.43...,  -0.90...),
 5: (0.97...,  -0.22...),
 6: (0.78...,   0.62...)}
sage: G.plot(layout="circular")
Graphics object consisting of 22 graphics primitives
```

layout_default(*by_component=True*, ***options*)

Return a spring layout for this graph.

INPUT:

- `by_components` – boolean (default: `True`);
- `**options` – options for method *spring_layout_fast()*

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: g = graphs.LadderGraph(3) #TODO!!!!
sage: g.layout_spring()
{0: [0.73..., -0.29...],
 1: [1.37..., 0.30...],
 2: [2.08..., 0.89...],
 3: [1.23..., -0.83...],
 4: [1.88..., -0.30...],
 5: [2.53..., 0.22...]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout="spring")
Graphics object consisting of 34 graphics primitives
```

layout_extend_randomly(*pos*, *dim=2*)

Extend randomly a partial layout

INPUT:

- `pos` – a dictionary mapping vertices to positions
- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3

OUTPUT: a dictionary mapping vertices to positions

The vertices not referenced in `pos` are assigned random positions within the box delimited by the other vertices.

EXAMPLES:

```
sage: H = digraphs.ButterflyGraph(1)
sage: pos = {('0', 0): (0, 0), ('1', 1): (1, 1)}
sage: H.layout_extend_randomly(pos)  # random
{('0', 0): (0, 0),
 ('0', 1): [0.0446..., 0.332...],
 ('1', 0): [0.1114..., 0.514...],
 ('1', 1): (1, 1)}
sage: xmin, xmax, ymin, ymax = H._layout_bounding_box(pos)
sage: (xmin, ymin) == (0, 0) and (xmax, ymax) == (1, 1)
True
```

layout_graphviz(*dim=2*, *prog='dot'*, ***options*)

Call `graphviz` to compute a layout of the vertices of this graph.

INPUT:

- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3
- `prog` – one of "dot", "neato", "twopi", "circo", or "fdp"
- `**options` – other parameters used by method *graphviz_string()*

EXAMPLES:

```
sage: g = digraphs.ButterflyGraph(2)
sage: g.layout_graphviz()  # optional - dot2tex graphviz
{('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...]}
sage: g.plot(layout="graphviz")  # optional - dot2tex graphviz
Graphics object consisting of 29 graphics primitives
```

Note: the actual coordinates are not deterministic

By default, an acyclic layout is computed using `graphviz`'s `dot` layout program. One may specify an alternative layout program:

```
sage: g.plot(layout = "graphviz", prog = "dot")   # optional - dot2tex␣
↪graphviz
Graphics object consisting of 29 graphics primitives
sage: g.plot(layout = "graphviz", prog = "neato") # optional - dot2tex␣
↪graphviz
Graphics object consisting of 29 graphics primitives
sage: g.plot(layout = "graphviz", prog = "twopi") # optional - dot2tex␣
↪graphviz
Graphics object consisting of 29 graphics primitives
sage: g.plot(layout = "graphviz", prog = "fdp")   # optional - dot2tex␣
↪graphviz
Graphics object consisting of 29 graphics primitives
sage: g = graphs.BalancedTree(5,2)
```

(continues on next page)

(continued from previous page)

```
sage: g.plot(layout = "graphviz", prog = "circo")  # optional - dot2tex␣
↪graphviz
Graphics object consisting of 62 graphics primitives
```

Todo: Put here some cool examples showcasing graphviz features.

This requires `graphviz` and the `dot2tex` spkg. Here are some installation tips:

- Install `graphviz` >= 2.14 so that the programs `dot`, `neato`, etc. are in your path. The graphviz suite can be download from http://graphviz.org.
- Install `dot2tex` with `sage -i dot2tex`

Todo: Use the graphviz functionality of Networkx 1.0 once it will be merged into Sage.

layout_planar(*set_embedding=False*, *on_embedding=None*, *external_face=None*, *test=False*, *circular=False*, ***options*)

Compute a planar layout of the graph using Schnyder's algorithm.

If `set_embedding` is set, a new combinatorial embedding is computed for the layout. Otherwise: if `on_embedding` is provided, then that combinatorial embedding is used for the layout. Otherwise: if a combinatorial embedding is set to the instance field variable of the graph (e.g. using `set_embedding()`), then that one is used, and if no combinatorial embedding is set, then one is computed.

If the graph is not planar, an error is raised.

INPUT:

- `set_embedding` – boolean (default: `False`); whether to set the instance field variable that contains a combinatorial embedding to the combinatorial embedding used for the planar layout (see `get_embedding()`)
- `on_embedding` – dictionary (default: `None`); provide a combinatorial embedding
- `external_face` – a pair (u, v) of vertices (default: `None`); the external face of the drawing is chosen in such a way that u and v are consecutive vertices in the clockwise traversal of the external face, in particular uv has to be an edge of the graph. If `external_face == None`, an arbitrary external face is chosen.
- `test` – boolean (default: `False`); whether to perform sanity tests along the way
- `circular` – ignored

EXAMPLES:

```
sage: g = graphs.PathGraph(10)
sage: g.layout(layout='planar', save_pos=True, test=True)
{0: [3, 2],
 1: [4, 3],
 2: [3, 4],
 3: [4, 4],
 4: [2, 6],
 5: [8, 1],
 6: [1, 7],
 7: [0, 8],
```

(continues on next page)

(continued from previous page)

```
 8: [1, 1],
 9: [1, 0]}
sage: g = graphs.BalancedTree(3, 4)
sage: pos = g.layout(layout='planar', save_pos=True, test=True)
sage: pos[0]
[2, 116]
sage: pos[120]
[3, 64]
sage: g = graphs.CycleGraph(7)
sage: g.layout(layout='planar', save_pos=True, test=True)
{0: [1, 4], 1: [5, 1], 2: [0, 5], 3: [1, 0], 4: [1, 2], 5: [2, 1], 6: [4, 1]}
sage: g = graphs.CompleteGraph(5)
sage: g.layout(layout='planar', save_pos=True, test=True, set_embedding=True)
Traceback (most recent call last):
...
ValueError: Complete graph is not a planar graph
```

Choose the external face of the drawing:

```
sage: g = graphs.CompleteGraph(4)
sage: g.layout(layout='planar', external_face=(0,1))
{0: [0, 2], 1: [2, 1], 2: [1, 0], 3: [1, 1]}
sage: g.layout(layout='planar', external_face=(3,1))
{0: [2, 1], 1: [0, 2], 2: [1, 1], 3: [1, 0]}
```

Choose the embedding:

sage: H = graphs.LadderGraph(4) sage: em = {0:[1,4], 4:[0,5], 1:[5,2,0], 5:[4,6,1], 2:[1,3,6], 6:[7,5,2], 3:[7,2], 7:[3,6]} sage: p = H.layout_planar(on_embedding=em) sage: p # random {2: [8.121320343559642, 1], 3: [2.1213203435596424, 6], 7: [3.1213203435596424, 0], 0: [5.121320343559642, 3], 1: [3.1213203435596424, 5], 4: [4.121320343559642, 3], 5: [4.121320343559642, 2], 6: [3.1213203435596424, 1], 9: [9.698670612749268, 1], 8: [8.698670612749268, 1], 10: [9.698670612749268, 0]}

layout_ranked(*heights=None*, *dim=2*, *spring=False*, ***options*)

Return a ranked layout for this graph

INPUT:

- `heights` – dictionary (default: `None`); a dictionary mapping heights to the list of vertices at this height
- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3
- `spring` – boolean (default: `False`);
- `**options` – options for method *spring_layout_fast()*

OUTPUT: a dictionary mapping vertices to positions

Returns a layout computed by randomly arranging the vertices along the given heights

EXAMPLES:

```
sage: g = graphs.LadderGraph(3)
sage: g.layout_ranked(heights={i: (i, i+3) for i in range(3)})
{0: [0.668..., 0],
 1: [0.667..., 1],
 2: [0.677..., 2],
```

(continues on next page)

(continued from previous page)

```
 3: [1.34..., 0],
 4: [1.33..., 1],
 5: [1.33..., 2]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout="ranked", heights={i: (i, i+7) for i in range(7)})
Graphics object consisting of 34 graphics primitives
```

layout_spring(*by_component=True*, ***options*)
Return a spring layout for this graph.

INPUT:

- `by_components` – boolean (default: `True`);
- `**options` – options for method *spring_layout_fast()*

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: g = graphs.LadderGraph(3) #TODO!!!!
sage: g.layout_spring()
{0: [0.73..., -0.29...],
 1: [1.37..., 0.30...],
 2: [2.08..., 0.89...],
 3: [1.23..., -0.83...],
 4: [1.88..., -0.30...],
 5: [2.53..., 0.22...]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout="spring")
Graphics object consisting of 34 graphics primitives
```

layout_tree(*tree_orientation='down'*, *tree_root=None*, *dim=2*, ***options*)
Return an ordered tree layout for this graph.

The graph must be a tree (no non-oriented cycles).

INPUT:

- `tree_root` – a vertex (default: `None`); the root vertex of the tree. By default (`None`) a vertex is chosen close to the center of the tree.
- `tree_orientation` – string (default: `'down'`); the direction in which the tree is growing, can be `'up'`, `'down'`, `'left'` or `'right'`
- `dim` – integer (default: `2`); the number of dimensions of the layout, 2 or 3
- `**options` – other parameters not used here

If the tree has been given a planar embedding (fixed circular order on the set of neighbors of every vertex) using `set_embedding`, the algorithm will create a layout that respects this embedding.

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: G = graphs.RandomTree(80)
sage: G.plot(layout="tree", tree_orientation="right")
Graphics object consisting of 160 graphics primitives

sage: T = graphs.RandomLobster(25, 0.3, 0.3)
```

(continues on next page)

(continued from previous page)

```
sage: T.show(layout='tree', tree_orientation='up')

sage: G = graphs.HoffmanSingletonGraph()
sage: T = Graph()
sage: T.add_edges(G.min_spanning_tree(starting_vertex=0))
sage: T.show(layout='tree', tree_root=0)

sage: G = graphs.BalancedTree(2, 2)
sage: G.layout_tree(tree_root=0)
{0: (1.5, 0),
 1: (2.5, -1),
 2: (0.5, -1),
 3: (3.0, -2),
 4: (2.0, -2),
 5: (1.0, -2),
 6: (0.0, -2)}

sage: G = graphs.BalancedTree(2, 4)
sage: G.plot(layout="tree", tree_root=0, tree_orientation="up")
Graphics object consisting of 62 graphics primitives
```

Using the embedding when it exists:

```
sage: T = Graph([(0, 1), (0, 6), (0, 3), (1, 2), (1, 5), (3, 4), (3, 7), (3,␣
↪8)])
sage: T.set_embedding({0: [1, 6, 3], 1: [2, 5, 0], 2: [1], 3: [4, 7, 8, 0],
....:     4: [3], 5: [1], 6: [0], 7: [3], 8: [3]})
sage: T.layout_tree()
{0: (2.166..., 0),
 1: (3.5, -1),
 2: (4.0, -2),
 3: (1.0, -1),
 4: (2.0, -2),
 5: (3.0, -2),
 6: (2.0, -1),
 7: (1.0, -2),
 8: (0.0, -2)}
sage: T.plot(layout="tree", tree_root=3)
Graphics object consisting of 18 graphics primitives
```

lex_BFS(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic breadth first search (LexBFS) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the

codes are updated.

Time complexity is $O(n+m)$ where n is the number of vertices and m is the number of edges.

See [CK2008] for more details on the algorithm.

See also:

- Wikipedia article Lexicographic_breadth-first_search
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex BFS is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_BFS()) == g.order()
True
```

Lex BFS ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4,␣
↪5), (5, 6)])
sage: g.lex_BFS()
[1, 2, 3, 5, 4, 6]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_BFS(initial_vertex=2)
[2, 3, 1]
```

For a Chordal Graph, a reversed Lex BFS is a Perfect Elimination Order:

```
sage: g = graphs.PathGraph(3).lexicographic_product(graphs.CompleteGraph(2))
sage: g.lex_BFS(reverse=True)  # py2
[(2, 0), (2, 1), (1, 1), (1, 0), (0, 0), (0, 1)]
sage: g.lex_BFS(reverse=True)  # py3
[(2, 1), (2, 0), (1, 1), (1, 0), (0, 1), (0, 0)]
```

And the vertices at the end of the tree of discovery are, for chordal graphs, simplicial vertices (their neighborhood is a complete graph):

```
sage: g = graphs.ClawGraph().lexicographic_product(graphs.CompleteGraph(2))
sage: v = g.lex_BFS()[-1]
sage: peo, tree = g.lex_BFS(initial_vertex = v,  tree=True)
sage: leaves = [v for v in tree if tree.in_degree(v) ==0]
sage: all(g.subgraph(g.neighbors(v)).is_clique() for v in leaves)
True
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
```

(continues on next page)

(continued from previous page)

```
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

lex_DFS(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic depth first search (LexDFS) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. Lex DFS differs from Lex BFS only in the way codes are updated after each iteration.

Time complexity is $O(n + m)$ where n is the number of vertices and m is the number of edges.

See [CK2008] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex DFS is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_DFS()) == g.order()
True
```

Lex DFS ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4,␣
↪5), (5, 6)])
sage: g.lex_DFS()
[1, 2, 3, 5, 6, 4]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_DFS(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

lex_DOWN(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic DOWN search (LexDOWN) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. During the i-th iteration of the algorithm $n-i$ is prepended to the codes of all neighbors of the selected vertex that are left in the graph.

Time complexity is $O(n+m)$ where n is the number of vertices and m is the number of edges.

See [Mil2017] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph

EXAMPLES:

A Lex DOWN is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_DOWN()) == g.order()
True
```

Lex DOWN ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4,␣
↪5), (5, 6)])
sage: g.lex_DOWN()
[1, 2, 3, 4, 6, 5]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_DOWN(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

lex_UP(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic UP search (LexUP) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. During the i-th iteration of the algorithm i is appended to the codes of all neighbors of the selected vertex that are left in the graph.

Time complexity is $O(n+m)$ where n is the number of vertices and m is the number of edges.

See [Mil2017] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex UP is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_UP()) == g.order()
True
```

Lex UP ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4,␣
↪5), (5, 6)])
sage: g.lex_UP()
[1, 2, 4, 5, 6, 3]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_UP(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

lexicographic_product(*other*)

Return the lexicographic product of `self` and `other`.

The lexicographic product of G and H is the graph L with vertex set $V(L) = V(G) \times V(H)$, and $((u, v), (w, x))$ is an edge iff :

- (u, w) is an edge of G, or
- $u = w$ and (v, x) is an edge of H.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: L = C.lexicographic_product(Z); L
Graph on 10 vertices
sage: L.plot() # long time
Graphics object consisting of 36 graphics primitives
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: L = D.lexicographic_product(P); L
Graph on 200 vertices
sage: L.plot() # long time
Graphics object consisting of 3501 graphics primitives
```

line_graph(*labels=True*)

Returns the line graph of the (di)graph.

INPUT:

- `labels` – boolean (default: `True`); whether edge labels should be taken in consideration. If `labels=True`, the vertices of the line graph will be triples `(u,v,label)`, and pairs of vertices otherwise.

The line graph of an undirected graph G is an undirected graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G. In other words, an edge in H represents a path of length 2 in G.

The line graph of a directed graph G is a directed graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G and the terminal vertex of e is the initial vertex of f. In other words, an edge in H represents a (directed) path of length 2 in G.

Note: As a *Graph* object only accepts hashable objects as vertices (and as the vertices of the line graph are the edges of the graph), this code will fail if edge labels are not hashable. You can also set the argument `labels=False` to ignore labels.

See also:

- The *line_graph* module.
- *line_graph_forbidden_subgraphs()* – the forbidden subgraphs of a line graph.
- *is_line_graph()* – tests whether a graph is a line graph.

EXAMPLES:

```
sage: g = graphs.CompleteGraph(4)
sage: h = g.line_graph()
sage: h.vertices()
[(0, 1, None),
(0, 2, None),
(0, 3, None),
(1, 2, None),
(1, 3, None),
(2, 3, None)]
sage: h.am()
[0 1 1 1 1 0]
[1 0 1 1 0 1]
[1 1 0 0 1 1]
[1 1 0 0 1 1]
[1 0 1 1 0 1]
[0 1 1 1 1 0]
sage: h2 = g.line_graph(labels=False)
sage: h2.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: h2.am() == h.am()
True
sage: g = DiGraph([[1..4], lambda i,j: i < j])
sage: h = g.line_graph()
sage: h.vertices()
[(1, 2, None),
(1, 3, None),
(1, 4, None),
(2, 3, None),
(2, 4, None),
(3, 4, None)]
sage: h.edges()
[((1, 2, None), (2, 3, None), None),
 ((1, 2, None), (2, 4, None), None),
 ((1, 3, None), (3, 4, None), None),
 ((2, 3, None), (3, 4, None), None)]
```

longest_path(*s=None*, *t=None*, *use_edge_labels=False*, *algorithm='MILP'*, *solver=None*, *verbose=0*)

Return a longest path of `self`.

INPUT:

- `s` – a vertex (default: `None`); forces the source of the path (the method then returns the longest path starting at `s`). The argument is set to `None` by default, which means that no constraint is set upon the first vertex in the path.
- `t` – a vertex (default: `None`); forces the destination of the path (the method then returns the longest path ending at `t`). The argument is set to `None` by default, which means that no constraint is set upon the last vertex in the path.
- `use_edge_labels` – boolean (default: `False`); whether to compute a path with maximum weight where the weight of an edge is defined by its label (a label set to `None` or `{}` being considered as a weight of 1), or to compute a path with the longest possible number of edges (i.e., edge weights are set to 1)
- `algorithm` – string (default: `"MILP"`); the algorithm to use among `"MILP"` and `"backtrack"`. Two remarks on this respect:
  - While the MILP formulation returns an exact answer, the backtrack algorithm is a randomized heuristic.
  - As the backtrack algorithm does not support edge weighting, setting `use_edge_labels=True` will force the use of the MILP algorithm.
- `solver` – string (default: `None`); specifies the Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: The length of a path is assumed to be the number of its edges, or the sum of their labels (when `use_edge_labels == True`).

OUTPUT:

A subgraph of `self` corresponding to a (directed if `self` is directed) longest path. If `use_edge_labels == True`, a pair `weight, path` is returned.

ALGORITHM:

Mixed Integer Linear Programming (this problem is known to be NP-Hard).

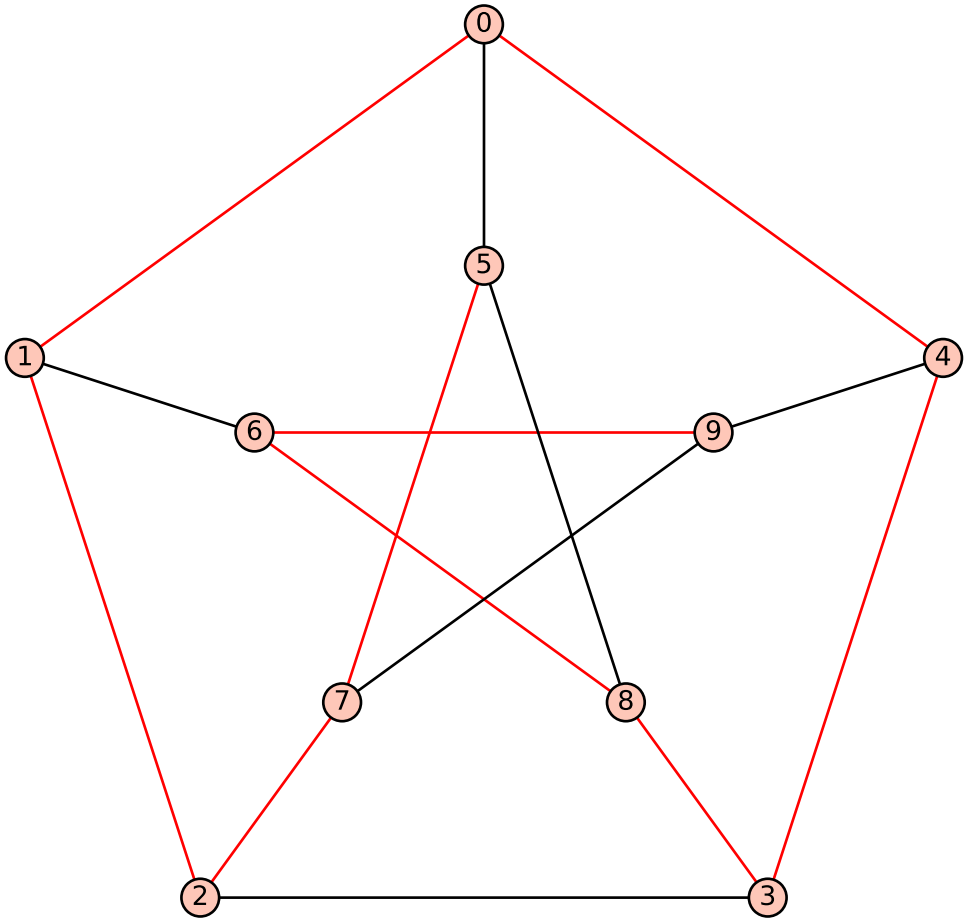
EXAMPLES:

Petersen's graph being hypohamiltonian, it has a longest path of length $n - 2$:

```
sage: g = graphs.PetersenGraph()
sage: lp = g.longest_path()
sage: lp.order() >= g.order() - 2
True
```

The heuristic totally agrees:

```
sage: g = graphs.PetersenGraph()
sage: g.longest_path(algorithm="backtrack").edges(labels=False)
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 9), (5, 7), (5, 8), (6, 8), (6, 9)]
```


0
1
2
3
4
5
6
7
8
9

Let us compute longest paths on random graphs with random weights. Each time, we ensure the resulting graph is indeed a path:

```
sage: for i in range(20):
....:     g = graphs.RandomGNP(15, 0.3)
....:     for u, v in g.edge_iterator(labels=False):
....:         g.set_edge_label(u, v, random())
....:     lp = g.longest_path()
....:     if (not lp.is_forest() or
....:         not max(lp.degree()) <= 2 or
....:         not lp.is_connected()):
....:         print("Error!")
....:         break
```

loop_edges(*labels=True*)

Return a list of all loops in the (di)graph

INPUT:

- `labels` – boolean (default: `True`); whether returned edges have labels (`(u,v,l)`) or not (`(u,v)`)

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
sage: G.loop_edges(labels=False)
[(0, 0), (1, 1), (2, 2), (3, 3)]
sage: G.allows_loops()
True
sage: G.has_loops()
True
sage: G.allow_loops(False)
sage: G.has_loops()
False
sage: G.loop_edges()
[]
sage: G.edges()
[(2, 3, None)]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0, 0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
```

(continues on next page)

(continued from previous page)

```
sage: D.has_loops()
False
sage: D.edges()
[]

sage: G = graphs.PetersenGraph()
sage: G.loops()
[]
```

```
sage: D = DiGraph(4, loops=True)
sage: D.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: D.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
```

```
sage: G = Graph(4, loops=True, multiedges=True, sparse=True)
sage: G.add_edges((i, i) for i in range(4))
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
sage: G.add_edges([(0, 0), (1, 1)])
sage: G.loop_edges(labels=False)
[(0, 0), (0, 0), (1, 1), (1, 1), (2, 2), (3, 3)]
```

loop_vertices()

Return a list of vertices with loops

EXAMPLES:

```
sage: G = Graph({0: [0], 1: [1, 2, 3], 2: [3]}, loops=True)
sage: G.loop_vertices()
[0, 1]
```

loops(*labels=True*)

Return a list of all loops in the (di)graph

INPUT:

- `labels` – boolean (default: `True`); whether returned edges have labels (`(u,v,l)`) or not (`(u,v)`)

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
sage: G.loop_edges(labels=False)
[(0, 0), (1, 1), (2, 2), (3, 3)]
sage: G.allows_loops()
True
sage: G.has_loops()
True
sage: G.allow_loops(False)
sage: G.has_loops()
```

(continues on next page)

(continued from previous page)

```
False
sage: G.loop_edges()
[]
sage: G.edges()
[(2, 3, None)]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0, 0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]

sage: G = graphs.PetersenGraph()
sage: G.loops()
[]
```

```
sage: D = DiGraph(4, loops=True)
sage: D.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: D.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
```

```
sage: G = Graph(4, loops=True, multiedges=True, sparse=True)
sage: G.add_edges((i, i) for i in range(4))
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
sage: G.add_edges([(0, 0), (1, 1)])
sage: G.loop_edges(labels=False)
[(0, 0), (0, 0), (1, 1), (1, 1), (2, 2), (3, 3)]
```

max_cut (*value_only=True*, *use_edge_labels=False*, *vertices=False*, *solver=None*, *verbose=0*)

Return a maximum edge cut of the graph.

For more information, see the Wikipedia article Maximum_cut.

INPUT:

- `value_only` – boolean (default: `False`); whether to return only the size of the maximum edge cut, or to also return the list of edges of the maximum edge cut
- `use_edge_labels` – boolean (default: `False`); whether to compute a weighted maximum cut where the weight of an edge is defined by its label (if an edge has no label, 1 is assumed), or to compute a cut of maximum cardinality (i.e., edge weights are set to 1)
- `vertices` – boolean (default: `False`); whether to return the two sets of vertices that are disconnected by the cut. This implies `value_only=False`.

- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

Quite obviously, the max cut of a bipartite graph is the number of edges, and the two sets of vertices are the two sides:

```
sage: g = graphs.CompleteBipartiteGraph(5,6)
sage: [ value, edges, [ setA, setB ]] = g.max_cut(vertices=True)
sage: value == 5*6
True
sage: bsetA, bsetB  = map(list,g.bipartite_sets())
sage: (bsetA == setA and bsetB == setB ) or ((bsetA == setB and bsetB == setA
↪))
True
```

The max cut of a Petersen graph:

```
sage: g=graphs.PetersenGraph()
sage: g.max_cut()
12
```

merge_vertices(*vertices*)

Merge vertices.

This function replaces a set S of vertices by a single vertex v_{new}, such that the edge uv_{new} exists if and only if $\exists v' \in S : (u, v') \in G$.

The new vertex is named after the first vertex in the list given in argument. If this first name is $None$, a new vertex is created.

In the case of multigraphs, the multiplicity is preserved.

INPUT:

- `vertices` – the list of vertices to be merged

Note: If `u` and `v` are distinct vertices in `vertices`, any edges between `u` and `v` will be lost.

EXAMPLES:

```
sage: g = graphs.CycleGraph(3)
sage: g.merge_vertices([0, 1])
sage: g.edges()
[(0, 2, None)]

sage: P = graphs.PetersenGraph()
sage: P.merge_vertices([5, 7])
sage: P.vertices()
[0, 1, 2, 3, 4, 5, 6, 8, 9]
```

When the first vertex in `vertices` is `None`, a new vertex is created:

```
sage: g = graphs.CycleGraph(5)
sage: g.vertices()
```

(continues on next page)

(continued from previous page)

```
[0, 1, 2, 3, 4]
sage: g.merge_vertices([None, 1, 3])
sage: g.edges(labels=False)
[(0, 4), (0, 5), (2, 5), (4, 5)]
```

With a Multigraph

```
sage: g = graphs.CycleGraph(3)
sage: g.allow_multiple_edges(True)
sage: g.merge_vertices([0, 1])
sage: g.edges(labels=False)
[(0, 2), (0, 2)]
```

min_spanning_tree(*weight_function=None*, *algorithm='Prim_Boost'*, *starting_vertex=None*, *check=False*, *by_weight=False*)

Return the edges of a minimum spanning tree.

At the moment, no algorithm for directed graph is implemented: if the graph is directed, a minimum spanning tree of the corresponding undirected graph is returned.

We expect all weights of the graph to be convertible to float. Otherwise, an exception is raised.

INPUT:

- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight. The `weight_function` can be used to transform the label into a weight (note that, if the weight returned is not convertible to a float, an error is raised)
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `"Prim_Boost"`); the algorithm to use in computing a minimum spanning tree of `G`. The following algorithms are supported:
  - `"Prim_Boost"` – Prim's algorithm (Boost implementation)
  - `"Prim_fringe"` – a variant of Prim's algorithm that ignores the labels on the edges
  - `"Prim_edge"` – a variant of Prim's algorithm
  - `"Kruskal"` – Kruskal's algorithm
  - `"Filter_Kruskal"` – a variant of Kruskal's algorithm [OSS2009]
  - `"Kruskal_Boost"` – Kruskal's algorithm (Boost implementation)
  - `"Boruvka"` – Boruvka's algorithm
  - `NetworkX` – uses NetworkX's minimum spanning tree implementation
- `starting_vertex` – a vertex (default: `None`); the vertex from which to begin the search for a minimum spanning tree (available only for `Prim_fringe` and `Prim_edge`).
- `check` – boolean (default: `False`); whether to first perform sanity checks on the input graph `G`. If appropriate, `check` is passed on to any minimum spanning tree functions that are invoked from the current method. See the documentation of the corresponding functions for details on what sort of sanity checks will be performed.

OUTPUT:

The edges of a minimum spanning tree of `G`, if one exists, otherwise returns the empty list.

See also:

- *sage.graphs.spanning_tree.kruskal()*
- *sage.graphs.spanning_tree.filter_kruskal()*
- *sage.graphs.spanning_tree.boruvka()*
- *sage.graphs.base.boost_graph.min_spanning_tree()*

EXAMPLES:

Kruskal's algorithm:

```
sage: g = graphs.CompleteGraph(5)
sage: len(g.min_spanning_tree())
4
sage: weight = lambda e: 1 / ((e[0] + 1) * (e[1] + 1))
sage: sorted(g.min_spanning_tree(weight_function=weight))
[(0, 4, None), (1, 4, None), (2, 4, None), (3, 4, None)]
sage: sorted(g.min_spanning_tree(weight_function=weight, algorithm='Kruskal_
↪Boost'))
[(0, 4, None), (1, 4, None), (2, 4, None), (3, 4, None)]
sage: g = graphs.PetersenGraph()
sage: g.allow_multiple_edges(True)
sage: g.add_edges(g.edge_iterator())
sage: sorted(g.min_spanning_tree())
[(0, 1, None), (0, 4, None), (0, 5, None), (1, 2, None), (1, 6, None), (3, 8,␣
↪None), (5, 7, None), (5, 8, None), (6, 9, None)]
```

Boruvka's algorithm:

```
sage: sorted(g.min_spanning_tree(algorithm='Boruvka'))
[(0, 1, None), (0, 4, None), (0, 5, None), (1, 2, None),  (1, 6, None), (2, 3,
↪ None), (2, 7, None),  (3, 8, None), (4, 9, None)]
```

Prim's algorithm:

```
sage: g = graphs.CompleteGraph(5)
sage: sorted(g.min_spanning_tree(algorithm='Prim_edge', starting_vertex=2,␣
↪weight_function=weight))
[(0, 4, None), (1, 4, None), (2, 4, None), (3, 4, None)]
sage: sorted(g.min_spanning_tree(algorithm='Prim_fringe', starting_vertex=2,␣
↪weight_function=weight))
[(0, 4, None), (1, 4, None), (2, 4, None), (3, 4, None)]
sage: sorted(g.min_spanning_tree(weight_function=weight, algorithm='Prim_Boost
↪'))
[(0, 4, None), (1, 4, None), (2, 4, None), (3, 4, None)]
```

NetworkX algorithm:

```
sage: sorted(g.min_spanning_tree(algorithm='NetworkX'))
[(0, 1, None), (0, 2, None), (0, 3, None), (0, 4, None)]
```

More complicated weights:

```
sage: G = Graph([(0,1,{'name':'a','weight':1}), (0,2,{'name':'b','weight':3}),
↪ (1,2,{'name':'b','weight':1})])
sage: sorted(G.min_spanning_tree(weight_function=lambda e: e[2]['weight']))
[(0, 1, {'name': 'a', 'weight': 1}), (1, 2, {'name': 'b', 'weight': 1})]
```

If the graph is not weighted, edge labels are not considered, even if they are numbers:

```
sage: g = Graph([(1, 2, 1), (1, 3, 2), (2, 3, 1)])
sage: sorted(g.min_spanning_tree())
[(1, 2, 1), (1, 3, 2)]
```

In order to use weights, we need either to set variable `weighted` to `True`, or to specify a weight function or set by_weight to `True`:

```
sage: g.weighted(True)
sage: sorted(g.min_spanning_tree())
[(1, 2, 1), (2, 3, 1)]
sage: g.weighted(False)
sage: sorted(g.min_spanning_tree())
[(1, 2, 1), (1, 3, 2)]
sage: sorted(g.min_spanning_tree(by_weight=True))
[(1, 2, 1), (2, 3, 1)]
sage: sorted(g.min_spanning_tree(weight_function=lambda e: e[2]))
[(1, 2, 1), (2, 3, 1)]
```

minimum_cycle_basis(*weight_function=None*, *by_weight=False*, *algorithm=None*)

Return a minimum weight cycle basis of the graph.

A cycle basis is a list of cycles (list of vertices forming a cycle) of `self`. Note that the vertices are not necessarily returned in the order in which they appear in the cycle.

A minimum weight cycle basis is a cycle basis that minimizes the sum of the weights (length for unweighted graphs) of its cycles.

Not implemented for directed graphs and multigraphs.

INPUT:

- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); algorithm to use:
  - If `algorithm = "NetworkX"`, use networkx implementation
  - If `algorithm = None`, use Sage Cython implementation

EXAMPLES:

```
sage: g = Graph([(1, 2, 3), (2, 3, 5), (3, 4, 8), (4, 1, 13), (1, 3, 250), (5,
↪ 6, 9), (6, 7, 17), (7, 5, 20)])
sage: sorted(g.minimum_cycle_basis(by_weight=True))
[[1, 2, 3], [1, 2, 3, 4], [5, 6, 7]]
sage: sorted(g.minimum_cycle_basis(by_weight=False))
[[1, 2, 3], [1, 3, 4], [5, 6, 7]]
sage: sorted(g.minimum_cycle_basis(by_weight=True, algorithm='NetworkX'))
```

(continues on next page)

(continued from previous page)

```
doctest:...: DeprecationWarning: connected_component_subgraphs is
deprecated and will be removedin 2.2. Use (G.subgraph(c).copy()
for c in connected_components(G))
[[1, 2, 3], [1, 2, 3, 4], [5, 6, 7]]
sage: g.minimum_cycle_basis(by_weight=False, algorithm='NetworkX')
[[1, 2, 3], [1, 3, 4], [5, 6, 7]]
```

```
sage: g = Graph([(1, 2), (2, 3), (3, 4), (4, 5), (5, 1), (5, 3)])
sage: sorted(g.minimum_cycle_basis(by_weight=False))
[[1, 2, 3, 5], [3, 4, 5]]
sage: sorted(g.minimum_cycle_basis(by_weight=False, algorithm='NetworkX'))
[[1, 2, 3, 5], [3, 4, 5]]
```

See also:

- *cycle_basis()*
- Wikipedia article Cycle_basis

multicommodity_flow(*terminals*, *integer=True*, *use_edge_labels=False*, *vertex_bound=False*, *solver=None*, *verbose=0*)

Solve a multicommodity flow problem.

In the multicommodity flow problem, we are given a set of pairs (s_i, t_i), called terminals meaning that s_i is willing some flow to t_i.

Even though it is a natural generalisation of the flow problem this version of it is NP-Complete to solve when the flows are required to be integer.

For more information, see the Wikipedia article Multi-commodity_flow_problem.

INPUT:

- `terminals` – a list of pairs (s_i, t_i) or triples (s_i, t_i, w_i) representing a flow from s_i to t_i of intensity w_i. When the pairs are of size 2, an intensity of 1 is assumed.
- `integer` boolean (default: `True`); whether to require an integer multicommodity flow
- `use_edge_labels` – boolean (default: `False`); whether to compute a multicommodity flow where each edge has a capacity defined by its label (if an edge has no label, capacity 1 is assumed), or to use default edge capacity of 1
- `vertex_bound` – boolean (default: `False`); whether to require that a vertex can stand at most 1 commodity of flow through it of intensity 1. Terminals can obviously still send or receive several units of flow even though vertex_bound is set to `True`, as this parameter is meant to represent topological properties.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default (quiet).

ALGORITHM:

(Mixed Integer) Linear Program, depending on the value of `integer`.

EXAMPLES:

An easy way to obtain a satisfiable multicommodity flow is to compute a matching in a graph, and to consider the paired vertices as terminals

```
sage: g = graphs.PetersenGraph()
sage: matching = [(u,v) for u,v,_ in g.matching()]
sage: h = g.multicommodity_flow(matching)
sage: len(h)
5
```

We could also have considered `g` as symmetric and computed the multicommodity flow in this version instead. In this case, however edges can be used in both directions at the same time:

```
sage: h = DiGraph(g).multicommodity_flow(matching)
sage: len(h)
5
```

An exception is raised when the problem has no solution

```
sage: h = g.multicommodity_flow([(u,v,3) for u,v in matching])
Traceback (most recent call last):
...
EmptySetError: the multicommodity flow problem has no solution
```

multiple_edges(*to_undirected=False*, *labels=True*, *sort=False*)

Return any multiple edges in the (di)graph.

INPUT:

- `to_undirected` – boolean (default `False`)
- `labels` – boolean (default `True`); whether to include labels
- `sort` - boolean (default `False`); whether to sort the result

EXAMPLES:

```
sage: G = Graph(multiedges=True, sparse=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0, 1)] * 3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges(sort=True)
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True, sparse=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0, 1)] * 3)
sage: D.has_multiple_edges()
```

(continues on next page)

(continued from previous page)

```
True
sage: D.multiple_edges(sort=True)
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]

sage: G = DiGraph({1: {2: 'h'}, 2: {1: 'g'}}, sparse=True)
sage: G.has_multiple_edges()
False
sage: G.has_multiple_edges(to_undirected=True)
True
sage: G.multiple_edges()
[]
sage: G.multiple_edges(to_undirected=True, sort=True)
[(1, 2, 'h'), (2, 1, 'g')]
```

multiway_cut(*vertices*, *value_only=False*, *use_edge_labels=False*, *solver=None*, *verbose=0*)

Return a minimum edge multiway cut.

A multiway cut for a vertex set S in a graph or a digraph G is a set C of edges such that any two vertices u, v in S are disconnected when removing the edges of C from G. (cf. http://www.d.kth.se/~viggo/wwwcompendium/node92.html)

Such a cut is said to be minimum when its cardinality (or weight) is minimum.

INPUT:

- `vertices` – iterable; the set of vertices
- `value_only` – boolean (default: `False`); whether to return only the size of the minimum multiway cut, or to return the list of edges of the multiway cut
- `use_edge_labels` – boolean (default: `False`); whether to compute a weighted minimum multiway cut where the weight of an edge is defined by its label (if an edge has no label, 1 is assumed), or to compute a cut of minimum cardinality (i.e., edge weights are set to 1)
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

Of course, a multiway cut between two vertices correspond to a minimum edge cut:

```
sage: g = graphs.PetersenGraph()
sage: g.edge_cut(0,3) == g.multiway_cut([0,3], value_only = True)
True
```

As Petersen's graph is 3-regular, a minimum multiway cut between three vertices contains at most 2×3 edges (which could correspond to the neighborhood of 2 vertices):

```
sage: g.multiway_cut([0,3,9], value_only = True) == 2*3
True
```

In this case, though, the vertices are an independent set. If we pick instead vertices 0, 9, and 7, we can save 4 edges in the multiway cut:

```
sage: g.multiway_cut([0,7,9], value_only = True) == 2*3 - 1
True
```

This example, though, does not work in the directed case anymore, as it is not possible in Petersen's graph to mutualise edges:

```
sage: g = DiGraph(g)
sage: g.multiway_cut([0,7,9], value_only = True) == 3*3
True
```

Of course, a multiway cut between the whole vertex set contains all the edges of the graph:

```
sage: C = g.multiway_cut(g.vertices())
sage: set(C) == set(g.edges())
True
```

name (*new=None*)

Return or set the graph's name.

INPUT:

- `new` – string (default: `None`); by default (`new == None`), the method returns the name of the graph. When `name` is set, the string representation of that object becomes the new name of the (di)graph (`new == ''` removes any name).

EXAMPLES:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7, 8], 6: [8,
↪9], 7: [9]}
sage: G = Graph(d); G
Graph on 10 vertices
sage: G.name("Petersen Graph"); G
Petersen Graph: Graph on 10 vertices
sage: G.name(new=""); G
Graph on 10 vertices
sage: G.name()
''
sage: G.name(42); G
42: Graph on 10 vertices
sage: G.name()
'42'
```

neighbor_iterator (*vertex*, *closed=False*)

Return an iterator over neighbors of `vertex`.

When `closed` is set to `True`, the returned iterator also contains `vertex`.

INPUT:

- `vertex` – a vertex of `self`
- `closed` – a boolean (default: `False`); whether to return the closed neighborhood of `vertex`, i.e., including `vertex`, or the open neighborhood in which `vertex` is included only if there is a loop on that vertex.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: for i in G.neighbor_iterator(0):
....:     print(i)
1
4
5
sage: D = G.to_directed()
sage: for i in D.neighbor_iterator(0):
....:     print(i)
1
4
5
```

```
sage: D = DiGraph({0: [1, 2], 3: [0]})
sage: list(D.neighbor_iterator(0))
[1, 2, 3]
```

```
sage: g = graphs.CubeGraph(3)
sage: sorted(list(g.neighbor_iterator('010', closed=True)))
['000', '010', '011', '110']
```

```
sage: g = Graph(3, loops = True)
sage: g.add_edge(0,1)
sage: g.add_edge(0,0)
sage: list(g.neighbor_iterator(0, closed=True))
[0, 1]
sage: list(g.neighbor_iterator(2, closed=True))
[2]
```

neighbors(*vertex*, *closed=False*)

Return a list of neighbors (in and out if directed) of `vertex`.

`G[vertex]` also works. When `closed` is set to `True`, the returned iterator also contains `vertex`.

INPUT:

- `vertex` – a vertex of `self`
- `closed` – a boolean (default: `False`); whether to return the closed neighborhood of `vertex`, i.e., including `vertex`, or the open neighborhood in which `vertex` is included only if there is a loop on that vertex.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: sorted(P.neighbors(3))
[2, 4, 8]
sage: sorted(P[4])
[0, 3, 9]
sage: sorted(P.neighbors(3, closed=True))
[2, 3, 4, 8]
```

networkx_graph(*weight_function=None*)

Return a new `NetworkX` graph from the Sage graph.

INPUT:

- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight.

EXAMPLES:

```
sage: G = graphs.TetrahedralGraph()
sage: N = G.networkx_graph()
sage: type(N)
<class 'networkx.classes.graph.Graph'>

sage: def weight_fn(e):
....:     return e[2]
sage: G1 = Graph([(1,2,1), (1,3,4), (2,3,3), (3,4,4)])
sage: H = G1.networkx_graph(weight_function=weight_fn)
sage: H.edges(data=True)
EdgeDataView([(1, 2, {'weight': 1}), (1, 3, {'weight': 4}), (2, 3, {'weight':␣
↪3}), (3, 4, {'weight': 4})])
sage: G2 = DiGraph([(1,2,1), (1,3,4), (2,3,3), (3,4,4), (3,4,5)],␣
↪multiedges=True)
sage: H = G2.networkx_graph(weight_function=weight_fn)
sage: H.edges(data=True)
OutMultiEdgeDataView([(1, 2, {'weight': 1}), (1, 3, {'weight': 4}), (2, 3, {
↪'weight': 3}), (3, 4, {'weight': 5}), (3, 4, {'weight': 4})])
```

nowhere_zero_flow (*k=None*, *solver=None*, *verbose=0*)

Return a `k`-nowhere zero flow of the (di)graph.

A flow on a graph $G = (V, E)$ is a pair (D, f) such that D is an orientation of G and f is a function on E satisfying

$$\sum_{u \in N_D^-(v)} f(uv) = \sum_{w \in N_D^+(v)} f(vw),\ \forall v \in V.$$

A `nowhere zero flow` on a graph $G = (V, E)$ is a flow (D, f) such that $f(e) \neq 0$ for every $e \in E$. For a positive integer k, a k-flow on a graph $G = (V, E)$ is a flow (D, f) such that $f : E \to Z$ and $-(k-1) \leq f(e) \leq k-1$ for every $e \in E$. A k-flow is positive if $f(e) > 0$ for every $e \in E$. A k-flow which is nowhere zero is called a *k-nowhere zero flow* (or k-NZF).

The following are equivalent.

- G admits a positive k-flow.
- G admits a k-NZF.
- Every orientation of G admits a k-NZF.

Furthermore, a (di)graph admits a k-NZF if and only if it is bridgeless and every bridgeless graph admits a 6-NZF [Sey1981]. See the Wikipedia article Nowhere-zero_flow for more details.

ALGORITHM:

If `self` is not directed, we search for a k-NZF on any orientation of `self` and then build a positive k-NZF by reverting edges with negative flow.

INPUT:

- `k` – integer (default: `6`); when set to a positive integer ≥ 2, search for a k-nowhere zero flow
- `solver` – (default: `None`); specifies a Linear Program solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity of the LP solver, where 0 means quiet.

OUTPUT:

A digraph with flow values stored as edge labels if a k-nowhere zero flow is found. If `self` is undirected, the edges of this digraph indicate the selected orientation. If no feasible solution is found, an error is raised.

EXAMPLES:

The Petersen graph admits a (positive) 5-nowhere zero flow, but no 4-nowhere zero flow:

```
sage: g = graphs.PetersenGraph()
sage: h = g.nowhere_zero_flow(k=5)
sage: sorted(set(h.edge_labels()))
[1, 2, 3, 4]
sage: h = g.nowhere_zero_flow(k=3)
Traceback (most recent call last):
...
EmptySetError: the problem has no feasible solution
```

The de Bruijn digraph admits a 2-nowhere zero flow:

```
sage: g = digraphs.DeBruijn(2, 3)
sage: h = g.nowhere_zero_flow(k=2)
sage: sorted(set(h.edge_labels()))
[-1, 1]
```

num_edges()

Return the number of edges.

Note that *num_edges()* also returns the number of edges in G.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

num_faces(*embedding=None*)

Return the number of faces of an embedded graph.

INPUT:

- `embedding` – dictionary (default: `None`); a combinatorial embedding dictionary. Format: `{v1: [v2,v3], v2: [v1], v3: [v1]}` (clockwise ordering of neighbors at each vertex). If set to `None` (default) the method will use the embedding stored as `self._embedding`. If none is stored, the method will compute the set of faces from the embedding returned by *is_planar()* (if the graph is, of course, planar).

EXAMPLES:

```
sage: T = graphs.TetrahedralGraph()
sage: T.num_faces()
4
```

num_verts()

Return the number of vertices.

Note that `len(G)` and *num_verts()* also return the number of vertices in G.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10
```

```
sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

number_of_loops()

Return the number of edges that are loops

EXAMPLES:

```
sage: G = Graph(4, loops=True)
sage: G.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: G.number_of_loops()
4
```

```
sage: D = DiGraph(4, loops=True)
sage: D.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: D.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: D.number_of_loops()
4
```

odd_girth(*algorithm='bfs'*, *certificate=False*)

Return the odd girth of the graph.

The odd girth is the length of the shortest cycle of odd length in the graph (directed cycle if the graph is directed). Bipartite graphs have infinite odd girth.

INPUT:

- `algorithm` – string (default: `"bfs"`); the algorithm to use:
  - `"bfs"` – BFS-based algorithm
  - any algorithm accepted by `charpoly()` for computation from the characteristic polynomial (see [Har1962] and [Big1993], p. 45)
- `certificate` – boolean (default: `False`); whether to return `(g, c)`, where `g` is the odd girth and `c` is a list of vertices of a (directed) cycle of length `g` in the graph, thus providing a certificate that the odd girth is at most `g`, or `None` if `g` is infinite. So far, this parameter is accepted only when `algorithm = "bfs"`.

EXAMPLES:

The McGee graph has girth 7 and therefore its odd girth is 7 as well:

```
sage: G = graphs.McGeeGraph()
sage: G.girth()
7
sage: G.odd_girth()
7
```

Any complete (directed) graph on more than 2 vertices contains a (directed) triangle and has thus odd girth 3:

```
sage: G = graphs.CompleteGraph(5)
sage: G.odd_girth(certificate=True)  # random
(3, [2, 1, 0])
```

(continues on next page)

(continued from previous page)

```
sage: G = digraphs.Complete(5)
sage: G.odd_girth(certificate=True)  # random
(3, [1, 2, 0])
```

Bipartite graphs have no odd cycle and consequently have infinite odd girth:

```
sage: G = graphs.RandomBipartite(6, 6, .5)
sage: G.odd_girth()
+Infinity
sage: G = graphs.Grid2dGraph(3, 4)
sage: G.odd_girth()
+Infinity
```

The odd girth of a (directed) graph with loops is 1:

```
sage: G = graphs.RandomGNP(10, .5)
sage: G.allow_loops(True)
sage: G.add_edge(0, 0)
sage: G.odd_girth()
1
sage: G = digraphs.RandomDirectedGNP(10, .5)
sage: G.allow_loops(True)
sage: G.add_edge(0, 0)
sage: G.odd_girth()
1
```

See also:

- *girth()* – return the girth of the graph.

order()

Return the number of vertices.

Note that `len(G)` and *num_verts()* also return the number of vertices in G.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10
```

```
sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

pagerank(*alpha=0.85*, *personalization=None*, *by_weight=False*, *weight_function=None*, *dangling=None*, *algorithm=None*)

Return the PageRank of the vertices of `self`.

PageRank is a centrality measure earlier used to rank web pages. The PageRank algorithm outputs the probability distribution that a random walker in the graph visits a vertex.

See the Wikipedia article PageRank for more information.

INPUT:

- `alpha` – float (default: `0.85`); damping parameter for PageRank. `alpha` is the click-through probability useful for preventing sinks. The probability at any step, that an imaginary surfer who is randomly clicking on links will continue is a damping factor d.
- `personalization` – dict (default: `None`); a dictionary keyed by vertices associating to each vertex a value. The personalization can be specified for a subset of the vertices, if not specified a nodes personalization value will be taken as zero. The sum of the values must be nonzero. By default (`None`), a uniform distribution is used.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `dangling` – dict (default: `None`); a dictionary keyed by a vertex the outedge of "dangling" vertices, (i.e., vertices without any outedges) points to and the dict value is the weight of that outedge. By default, dangling vertices are given outedges according to the personalization vector (uniform if not specified). It may be common to have the dangling dict to be the same as the personalization dict.
- **`algorithm` – string (default: `None`); the algorithm to use in** computing PageRank of `G`. The following algorithms are supported:
  - `NetworkX` – uses NetworkX's PageRank algorithm implementation Note that `'networkx'` does not support multigraphs.
  - `"Numpy"` – uses Numpy's PageRank algorithm implementation
  - `"Scipy"` – uses Scipy's PageRank algorithm implementation
  - `"igraph"` – uses igraph's PageRank algorithm implementation
  - `"None"` – uses best implementation available

OUTPUT: a dictionary containing the PageRank value of each node

---

Note: Parameters `alpha`, `by_weight` and `weight_function` are common to all algorithms. Parameters `personalization` and `dangling` are used only by algorithms `NetworkX`, `Numpy` and `Scipy`.

EXAMPLES:

```
sage: G = graphs.CycleGraph(4)
sage: G.pagerank(algorithm="Networkx")
{0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
sage: G.pagerank(alpha=0.50, algorithm="igraph")  # optional - python_igraph
↪# abs tol 1e-9
{0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25}
sage: G = Graph([(1, 2, 40), (2, 3, 50), (3, 4, 60), (1, 4, 70), (4, 5, 80),␣
↪(5, 6, 20)])
sage: G.pagerank(algorithm="NetworkX") # abs tol 1e-9
{1: 0.16112205885619568,
 2: 0.16195310432472196,
 3: 0.16112205885619568,
 4: 0.2375,
 5: 0.17775588228760858,
 6: 0.10054689567527803}
```

(continues on next page)

(continued from previous page)

```
sage: G.pagerank(algorithm="NetworkX", by_weight=True) # abs tol 1e-9
{1: 0.16459583718588988,
 2: 0.1397792859515451,
 3: 0.165398401843396,
 4: 0.3063198690713852,
 5: 0.17000576097071404,
 6: 0.053900844977069616}
sage: G.pagerank(algorithm="Numpy") # abs tol 1e-9
{1: 0.16112198303979114,
 2: 0.16195368558382248,
 3: 0.16112198303979122,
 4: 0.2375000000000002,
 5: 0.17775603392041756,
 6: 0.10054631441617742}
sage: G.pagerank(algorithm="Numpy", by_weight=True) # abs tol 1e-9
{1: 0.16459613361799788,
 2: 0.13977926864974763,
 3: 0.1653988472578896,
 4: 0.3063198780991534,
 5: 0.17000501912411242,
 6: 0.053900853251099105}
sage: G.pagerank(algorithm="Scipy") # abs tol 1e-9
{1: 0.16112205885619563,
 2: 0.1619531043247219,
 3: 0.16112205885619563,
 4: 0.2374999999999999,
 5: 0.17775588228760858,
 6: 0.100546895675278}
sage: G.pagerank(algorithm="Scipy", by_weight=True) # abs tol 1e-9
{1: 0.16459583718588994,
 2: 0.13977928595154515,
 3: 0.16539840184339605,
 4: 0.3063198690713853,
 5: 0.17000576097071141,
 6: 0.05390084497706962}
sage: G.pagerank(algorithm="igraph")  # optional - python_igraph # abs tol 1e-
↪9
{1: 0.16112198303979128,
 2: 0.16195368558382262,
 3: 0.16112198303979125,
 4: 0.23749999999999993,
 5: 0.17775603392041744,
 6: 0.10054631441617742}
sage: G.pagerank() # abs tol 1e-9
{1: 0.16112198303979114,
 2: 0.16195368558382248,
 3: 0.16112198303979122,
 4: 0.2375000000000002,
 5: 0.17775603392041756,
 6: 0.10054631441617742}
sage: G.pagerank(by_weight=True) # abs tol 1e-9
{1: 0.16459613361799788,
 2: 0.13977926864974763,
 3: 0.1653988472578896,
 4: 0.3063198780991534,
 5: 0.17000501912411242,
 6: 0.053900853251099105}
```

See also:

- Wikipedia article PageRank

periphery(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)
Return the set of vertices in the periphery of the (di)graph.

The periphery is the set of vertices whose eccentricity is equal to the diameter of the (di)graph, i.e., achieving the maximum eccentricity.

For more information and examples on how to use input variables, see *shortest_paths()* and *eccentricity()*

INPUT:

- `by_weight` – boolean (default: `False`); if `True`, edge weights are taken into account; if False, all edges have weight 1
- `algorithm` – string (default: `None`); see method *eccentricity()* for the list of available algorithms
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); if `True`, we check that the `weight_function` outputs a number for each edge

EXAMPLES:

```
sage: G = graphs.DiamondGraph()
sage: G.periphery()
[0, 3]
sage: P = graphs.PetersenGraph()
sage: P.subgraph(P.periphery()) == P
True
sage: S = graphs.StarGraph(19)
sage: S.periphery()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
sage: G = Graph()
sage: G.periphery()
[]
sage: G.add_vertex()
0
sage: G.periphery()
[0]
```

planar_dual(*embedding=None*)
Return the planar dual of an embedded graph.

A combinatorial embedding of a graph is a clockwise ordering of the neighbors of each vertex. From this information one can obtain the dual of a plane graph, which is what the method returns. The vertices of the dual graph correspond to faces of the primal graph.

INPUT:

- `embedding` – dictionary (default: `None`); a combinatorial embedding dictionary. Format: `{v1: [v2,v3], v2: [v1], v3: [v1]}` (clockwise ordering of neighbors at each vertex). If set to `None` (default) the method will use the embedding stored as `self._embedding`. If none is stored, the method will compute the set of faces from the embedding returned by *is_planar()* (if the graph is, of course, planar).

EXAMPLES:

```
sage: C = graphs.CubeGraph(3)
sage: C.planar_dual()
Graph on 6 vertices
sage: graphs.IcosahedralGraph().planar_dual().is_isomorphic(graphs.
↪DodecahedralGraph())
True
```

The planar dual of the planar dual is isomorphic to the graph itself:

```
sage: g = graphs.BuckyBall()
sage: g.planar_dual().planar_dual().is_isomorphic(g)
True
```

See also:

- *faces()*
- *set_embedding()*
- *get_embedding()*
- *is_planar()*

Todo: Implement the method for graphs that are not 3-vertex-connected, or at least have a faster 3-vertex-connectivity test (trac ticket #24635).

---

plot(***options*)

Return a Graphics object representing the (di)graph.

INPUT:

- pos – an optional positioning dictionary
- layout – string (default: None); specifies a kind of layout to use, takes precedence over pos
  - 'circular' – plots the graph with vertices evenly distributed on a circle
  - 'spring' – uses the traditional spring layout, using the graph's current positions as initial positions
  - 'tree' – the (di)graph must be a tree. One can specify the root of the tree using the keyword tree_root, otherwise a root will be selected at random. Then the tree will be plotted in levels, depending on minimum distance for the root.
- vertex_labels – boolean (default: True); whether to print vertex labels
- edge_labels – boolean (default: False); whether to print edge labels. If True, the result of str(l) is printed on the edge for each label l. Labels equal to None are not printed (to set edge labels, see *set_edge_label()*).
- edge_labels_background – the color of the edge labels background. The default is "white". To achieve a transparent background use "transparent".
- vertex_size – size of vertices displayed
- vertex_shape – the shape to draw the vertices, for example "o" for circle or "s" for square. Whole list is available at https://matplotlib.org/api/markers_api.html. (Not available for multiedge digraphs.)

- graph_border – boolean (default: False); whether to include a box around the graph
- vertex_colors – dictionary (default: None); optional dictionary to specify vertex colors: each key is a color recognizable by matplotlib, and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it doesn't get drawn).
- edge_colors – dictionary (default: None); a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges.
- partition – a partition of the vertex set (default: None); if specified, plot will show each cell in a different color. vertex_colors takes precedence.
- talk – boolean (default: False); if True, prints large vertices with white backgrounds so that labels are legible on slides
- iterations – integer; how many iterations of the spring layout algorithm to go through, if applicable
- color_by_label – a boolean or dictionary or function (default: False); whether to color each edge with a different color according to its label; the colors are chosen along a rainbow, unless they are specified by a function or dictionary mapping labels to colors; this option is incompatible with edge_color and edge_colors.
- heights – dictionary (default: None); if specified, this is a dictionary from a set of floating point heights to a set of vertices
- edge_style – keyword arguments passed into the edge-drawing routine. This currently only works for directed graphs, since we pass off the undirected graph to networkx
- tree_root – a vertex (default: None); if specified, this vertex is used as the root for the layout="tree" option. Otherwise, then one is chosen at random. Ignored unless layout='tree'.
- tree_orientation – string (default: "down"); one of "up" or "down". If "up" (resp., "down"), then the root of the tree will appear on the bottom (resp., top) and the tree will grow upwards (resp. downwards). Ignored unless layout='tree'.
- save_pos – boolean (default: False); save position computed during plotting

---

Note:

- This method supports any parameter accepted by sage.plot.graphics.Graphics.show().
- See the documentation of the *sage.graphs.graph_plot* module for information and examples of how to define parameters that will be applied to **all** graph plots.
- Default parameters for this method *and a specific graph* can also be set through the options mechanism. For more information on this different way to set default parameters, see the help of the options decorator.
- See also the *sage.graphs.graph_latex* module for ways to use LaTeX to produce an image of a graph.

EXAMPLES:

```
sage: from sage.graphs.graph_plot import graphplot_options
sage: sorted(graphplot_options.items())
[...]

sage: from math import sin, cos, pi
```

(continues on next page)

(continued from previous page)

```
sage: P = graphs.PetersenGraph()
sage: d = {'#FF0000': [0, 5], '#FF9900': [1, 6], '#FFFF00': [2, 7], '#00FF00
↪': [3, 8], '#0000FF': [4, 9]}
sage: pos_dict = {}
sage: for i in range(5):
....:  x = float(cos(pi/2 + ((2*pi)/5)*i))
....:  y = float(sin(pi/2 + ((2*pi)/5)*i))
....:  pos_dict[i] = [x,y]
sage: for i in range(5, 10):
....:  x = float(0.5*cos(pi/2 + ((2*pi)/5)*i))
....:  y = float(0.5*sin(pi/2 + ((2*pi)/5)*i))
....:  pos_dict[i] = [x,y]
sage: pl = P.plot(pos=pos_dict, vertex_colors=d)
sage: pl.show()
```

```
sage: C = graphs.CubeGraph(8)
sage: P = C.plot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

```
sage: G = graphs.HeawoodGraph()
sage: for u, v, l in G.edges(sort=False):
....:     G.set_edge_label(u, v, '(' + str(u) + ',' + str(v) + ')')
sage: G.plot(edge_labels=True).show()
```

```
sage: D = DiGraph( { 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4], 4: [17,
↪ 5], 5: [6, 15], 6: [7], 7: [8, 14], 8: [9], 9: [10, 13], 10: [11], 11: [12,
↪ 18], 12: [16, 13], 13: [14], 14: [15], 15: [16], 16: [17], 17: [18], 18:␣
↪[19], 19: []} , sparse=True)
sage: for u,v,l in D.edges(sort=False):
....:     D.set_edge_label(u, v, '(' + str(u) + ',' + str(v) + ')')
sage: D.plot(edge_labels=True, layout='circular').show()
```

```
sage: from sage.plot.colors import rainbow
sage: C = graphs.CubeGraph(5)
sage: R = rainbow(5)
sage: edge_colors = {R[i]: [] for i in range(5)}
sage: for u, v, l in C.edges(sort=False):
....:  for i in range(5):
....:      if u[i] != v[i]:
....:          edge_colors[R[i]].append((u, v, l))
sage: C.plot(vertex_labels=False, vertex_size=0, edge_colors=edge_colors).
↪show()
```

```
sage: D = graphs.DodecahedralGraph()
sage: Pi = [[6,5,15,14,7], [16,13,8,2,4], [12,17,9,3,1], [0,19,18,10,11]]
sage: D.show(partition=Pi)
```

```
sage: G = graphs.PetersenGraph()
sage: G.allow_loops(True)
sage: G.add_edge(0, 0)
sage: G.show()
```

```
sage: D = DiGraph({0: [0, 1], 1: [2], 2: [3]}, loops=True)
sage: D.show()
```

(continues on next page)

(continued from previous page)

```
sage: D.show(edge_colors={(0, 1, 0): [(0, 1, None), (1, 2, None)], (0, 0, 0):␣
↪[(2, 3, None)]})
```

```
sage: pos = {0: [0.0, 1.5], 1: [-0.8, 0.3], 2: [-0.6, -0.8], 3: [0.6, -0.8],␣
↪4: [0.8, 0.3]}
sage: g = Graph({0: [1], 1: [2], 2: [3], 3: [4], 4: [0]})
sage: g.plot(pos=pos, layout='spring', iterations=0)
Graphics object consisting of 11 graphics primitives
```

```
sage: G = Graph()
sage: P = G.plot()
sage: P.axes()
False
sage: G = DiGraph()
sage: P = G.plot()
sage: P.axes()
False
```

```
sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: (0.0..., 1.0...),
 1: (-0.95..., 0.30...),
 2: (-0.58..., -0.80...),
 3: (0.58..., -0.80...),
 4: (0.95..., 0.30...),
 5: (0.0..., 0.5...),
 6: (-0.47..., 0.15...),
 7: (-0.29..., -0.40...),
 8: (0.29..., -0.40...),
 9: (0.47..., 0.15...)}
sage: P = G.plot(save_pos=True, layout='spring')

The following illustrates the format of a position dictionary.

sage: G.get_pos() # currently random across platforms, see #9593
{0: [1.17..., -0.855...],
 1: [1.81..., -0.0990...],
 2: [1.35..., 0.184...],
 3: [1.51..., 0.644...],
 4: [2.00..., -0.507...],
 5: [0.597..., -0.236...],
 6: [2.04..., 0.687...],
 7: [1.46..., -0.473...],
 8: [0.902..., 0.773...],
 9: [2.48..., -0.119...]}
```

```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0: [0], 1: [4, 5, 1], 2: [2], 3: [3, 6]})
Graphics object consisting of 14 graphics primitives
```


```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0: [0], 1: [4, 5, 1], 2: [2], 3: [3, 6]})
Graphics object consisting of 14 graphics primitives
```


(continues on next page)

(continued from previous page)

```
sage: t.set_edge_label(0, 1, -7)
sage: t.set_edge_label(0, 5, 3)
sage: t.set_edge_label(0, 5, 99)
sage: t.set_edge_label(1, 2, 1000)
sage: t.set_edge_label(3, 2, 'spam')
sage: t.set_edge_label(2, 6, 3/2)
sage: t.set_edge_label(0, 4, 66)
sage: t.plot(heights={0: [0], 1: [4, 5, 1], 2: [2], 3: [3, 6]}, edge_
↪labels=True)
Graphics object consisting of 20 graphics primitives
```

```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(layout='tree')
Graphics object consisting of 14 graphics primitives
```

```
sage: t = DiGraph('JCC???@A??GO??CO??GO??')
sage: t.plot(layout='tree', tree_root=0, tree_orientation="up")
Graphics object consisting of 22 graphics primitives
sage: D = DiGraph({0: [1, 2, 3], 2: [1, 4], 3: [0]})
sage: D.plot()
Graphics object consisting of 16 graphics primitives

sage: D = DiGraph(multiedges=True,sparse=True)
sage: for i in range(5):
....:     D.add_edge((i, i + 1, 'a'))
....:     D.add_edge((i, i - 1, 'b'))
sage: D.plot(edge_labels=True, edge_colors=D._color_by_label())
Graphics object consisting of 34 graphics primitives
sage: D.plot(edge_labels=True, color_by_label={'a': 'blue', 'b': 'red'}, edge_
↪style='dashed')
Graphics object consisting of 34 graphics primitives

sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0, 0, 'a'), (0, 0, 'b'), (0, 1, 'c'), (0, 1, 'd'),
....:     (0, 1, 'e'), (0, 1, 'f'), (0, 1, 'f'), (2, 1, 'g'), (2, 2, 'h')])
sage: g.plot(edge_labels=True, color_by_label=True, edge_style='dashed')
Graphics object consisting of 26 graphics primitives
```

```
sage: S = SupersingularModule(389)
sage: H = S.hecke_matrix(2)
sage: D = DiGraph(H,sparse=True)
sage: P = D.plot()
```

```
sage: G=Graph({'a':['a','b','b','b','e'],'b':['c','d','e'],'c':['c','d','d','d
↪'],'d':['e']}, sparse=True)
sage: G.show(pos={'a':[0,1],'b':[1,1],'c':[2,0],'d':[1,0],'e':[0,0]})
```

plot3d(*bgcolor=(1, 1, 1)*, *vertex_colors=None*, *vertex_size=0.06*, *vertex_labels=False*, *edge_colors=None*, *edge_size=0.02*, *edge_size2=0.0325*, *pos3d=None*, *color_by_label=False*, *engine='threejs'*, ***kwds*)

Plot a graph in three dimensions.

See also the *sage.graphs.graph_latex* module for ways to use LaTeX to produce an image of a graph.

INPUT:

- bgcolor – rgb tuple (default: (1,1,1))
- vertex_size – float (default: 0.06)
- vertex_labels – a boolean (default: False); whether to display vertices using text labels instead of spheres
- vertex_colors – dictionary (default: None); optional dictionary to specify vertex colors: each key is a color recognizable by tachyon (rgb tuple (default: (1,0,0))), and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it does not get drawn).
- edge_colors – dictionary (default: None); a dictionary specifying edge colors: each key is a color recognized by tachyon (default: (0,0,0)), and each entry is a list of edges.
- color_by_label – a boolean or dictionary or function (default: False) whether to color each edge with a different color according to its label; the colors are chosen along a rainbow, unless they are specified by a function or dictionary mapping labels to colors; this option is incompatible with edge_color and edge_colors.
- edge_size – float (default: 0.02)
- edge_size2 – float (default: 0.0325); used for Tachyon sleeves
- pos3d – a position dictionary for the vertices
- layout, iterations, ... – layout options; see *layout()*
- engine – string (default: 'threejs'); the renderer to use among:
  - 'threejs': interactive web-based 3D viewer using JavaScript and a WebGL renderer
  - 'jmol': interactive 3D viewer using Java
  - 'tachyon': ray tracer generating a static PNG image
- xres – resolution
- yres – resolution
- **kwds – passed on to the rendering engine

EXAMPLES:

```
sage: G = graphs.CubeGraph(5)
sage: G.plot3d(iterations=500, edge_size=None, vertex_size=0.04) # long time
Graphics3d Object
```

We plot a fairly complicated Cayley graph:

```
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.plot3d(vertex_size=0.03, edge_size=0.01, vertex_colors={(1,1,1):␣
↪list(G)}, bgcolor=(0,0,0), color_by_label=True, iterations=200) # long time
Graphics3d Object
```

Some Tachyon examples:

```
sage: D = graphs.DodecahedralGraph()
sage: P3D = D.plot3d(engine='tachyon')
sage: P3D.show() # long time
```

```
sage: G = graphs.PetersenGraph()
sage: G.plot3d(engine='tachyon', vertex_colors={(0,0,1): list(G)}).show() #␣
↪long time
```

```
sage: C = graphs.CubeGraph(4)
sage: C.plot3d(engine='tachyon', edge_colors={(0,1,0): C.edges(sort=False)},␣
↪vertex_colors={(1,1,1): list(C)}, bgcolor=(0,0,0)).show() # long time
```

```
sage: K = graphs.CompleteGraph(3)
sage: K.plot3d(engine='tachyon', edge_colors={(1,0,0): [(0,1,None)], (0,1,0):␣
↪[(0,2,None)], (0,0,1): [(1,2,None)]}).show() # long time
```

A directed version of the dodecahedron

```
sage: D = DiGraph({0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4], 4: [17,␣
↪5], 5: [6, 15], 6: [7], 7: [8, 14], 8: [9], 9: [10, 13], 10: [11], 11: [12,␣
↪18], 12: [16, 13], 13: [14], 14: [15], 15: [16], 16: [17], 17: [18], 18:␣
↪[19], 19: []})
sage: D.plot3d().show() # long time
```

```
sage: P = graphs.PetersenGraph().to_directed()
sage: from sage.plot.colors import rainbow
sage: R = rainbow(P.size(), 'rgbtuple')
sage: edge_colors = {R[i]: [e] for i, e in enumerate(P.edge_iterator())}
sage: P.plot3d(engine='tachyon', edge_colors=edge_colors).show() # long time
```

```
sage: G=Graph({'a':['a','b','b','b','e'],'b':['c','d','e'],'c':['c','d','d','d
↪'],'d':['e']},sparse=True)
sage: G.show3d()
Traceback (most recent call last):
...
NotImplementedError: 3D plotting of multiple edges or loops not implemented
```

Using the `partition` keyword:

```
sage: G = graphs.WheelGraph(7)
sage: G.plot3d(partition=[[0], [1, 2, 3, 4, 5, 6]])
Graphics3d Object
```

See also:

- *plot()*
- *graphviz_string()*

radius (*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the radius of the (di)graph.

The radius is defined to be the minimum eccentricity of any vertex, where the eccentricity is the maximum distance to any other vertex. For more information and examples on how to use input variables, see *shortest_paths()* and *eccentricity()*

INPUT:

- `by_weight` – boolean (default: `False`); if `True`, edge weights are taken into account; if False, all edges have weight 1

- algorithm – string (default: None); see method *eccentricity()* for the list of available algorithms
- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

The more symmetric a graph is, the smaller (diameter - radius) is:

```
sage: G = graphs.BarbellGraph(9, 3)
sage: G.radius()
3
sage: G.diameter()
6
```

```
sage: G = graphs.OctahedralGraph()
sage: G.radius()
2
sage: G.diameter()
2
```

random_edge(***kwds*)

Return a random edge of self.

INPUT:

- **kwds – arguments to be passed down to the *edge_iterator()* method

EXAMPLES:

The returned value is an edge of self:

```
sage: g = graphs.PetersenGraph()
sage: u,v = g.random_edge(labels=False)
sage: g.has_edge(u,v)
True
```

As the *edges()* method would, this function returns by default a triple (u, v, l) of values, in which l is the label of edge (u, v):

```
sage: g.random_edge()
(3, 4, None)
```

random_edge_iterator(**args*, ***kwds*)

Return an iterator over random edges of self.

The returned iterator enables to amortize the cost of accessing random edges, as can be done with multiple calls to method *random_edge()*.

INPUT:

- *args and **kwds – arguments to be passed down to the *edge_iterator()* method.

EXAMPLES:

The returned value is an iterator over the edges of self:

```
sage: g = graphs.PetersenGraph()
sage: it = g.random_edge_iterator()
sage: [g.has_edge(next(it)) for _ in range(5)]
[True, True, True, True, True]
```

As the *edges()* method would, this function returns by default a triple `(u, v, l)` of values, in which `l` is the label of edge `(u,v)`:

```
sage: print(next(g.random_edge_iterator())) # random
(0, 5, None)
sage: print(next(g.random_edge_iterator(labels=False))) # random
(5, 7)
```

random_subgraph(*p*, *inplace=False*)

Return a random subgraph containing each vertex with probability `p`.

INPUT:

- `p` – the probability of choosing a vertex
- `inplace` – boolean (default: `False`); using `inplace=True` will simply delete the extra vertices and edges from the current graph. This will modify the graph.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.random_subgraph(.25)
Subgraph of (Petersen graph): Graph on 4 vertices
```

random_vertex(***kwds*)

Return a random vertex of `self`.

INPUT:

- `**kwds` – arguments to be passed down to the *vertex_iterator()* method

EXAMPLES:

The returned value is a vertex of self:

```
sage: g = graphs.PetersenGraph()
sage: v = g.random_vertex()
sage: v in g
True
```

random_vertex_iterator(**args*, ***kwds*)

Return an iterator over random vertices of `self`.

The returned iterator enables to amortize the cost of accessing random vertices, as can be done with multiple calls to method *random_vertex()*.

INPUT:

- `*args` and `**kwds` – arguments to be passed down to the *vertex_iterator()* method

EXAMPLES:

The returned value is an iterator over the vertices of `self`:

```
sage: g = graphs.PetersenGraph()
sage: it = g.random_vertex_iterator()
```

(continues on next page)

(continued from previous page)

```
sage: [next(it) in g for _ in range(5)]
[True, True, True, True, True]
```

relabel(*perm=None*, *inplace=True*, *return_map=False*, *check_input=True*, *complete_partial_function=True*, *immutable=None*)

Relabels the vertices of `self`

INPUT:

- `perm` – a function, dictionary, iterable, permutation, or `None` (default: `None`)
- `inplace` – a boolean (default: `True`)
- `return_map` – a boolean (default: `False`)
- `check_input` (boolean) – whether to test input for correctness. *This can potentially be very time-consuming !*.
- `complete_partial_function` (boolean) – whether to automatically complete the permutation if some elements of the graph are not associated with any new name. In this case, those elements are not relabeled *This can potentially be very time-consuming !*.
- `immutable` (boolean) – with `inplace=False`, whether to create a mutable/immutable relabelled copy. `immutable=None` (default) means that the graph and its copy will behave the same way.

If `perm` is a function `f`, then each vertex `v` is relabeled to `f(v)`.

If `perm` is a dictionary `d`, then each vertex `v` (which should be a key of `d`) is relabeled to `d[v]`.

If `perm` is a list (or more generally, any iterable) of length `n`, then the first vertex returned by `G.vertices()` is relabeled to `l[0]`, the second to `l[1]`, ...

If `perm` is a permutation, then each vertex `v` is relabeled to `perm(v)`. Caveat: this assumes that the vertices are labelled $\{0, 1, ..., n-1\}$; since permutations act by default on the set $\{1, 2, ..., n\}$, this is achieved by identifying n and 0.

If `perm` is `None`, the graph is relabeled to be on the vertices $\{0, 1, ..., n-1\}$. This is *not* any kind of canonical labeling, but it is consistent (relabeling twice will give the same result).

If `inplace` is `True`, the graph is modified in place and `None` is returned. Otherwise a relabeled copy of the graph is returned.

If `return_map` is `True` a dictionary representing the relabelling map is returned (incompatible with `inplace==False`).

EXAMPLES:

```
sage: G = graphs.PathGraph(3)
sage: G.am()
[0 1 0]
[1 0 1]
[0 1 0]
```

Relabeling using a dictionary. Note that the dictionary does not define the new label of vertex 0:

```
sage: G.relabel({1:2,2:1}, inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

This is because the method automatically "extends" the relabeling to the missing vertices (whose label will not change). Checking that all vertices have an image can require some time, and this feature can be disabled (at your own risk):

```
sage: G.relabel({1:2,2:1}, inplace=False, complete_partial_function = False).
↪am()
Traceback (most recent call last):
...
KeyError: 0
```

Relabeling using a list:

```
sage: G.relabel([0,2,1], inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

Relabeling using an iterable:

```
sage: G.relabel(iter((0,2,1)), inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

Relabeling using a Sage permutation:

```
sage: G = graphs.PathGraph(3)
sage: from sage.groups.perm_gps.permgroup_named import SymmetricGroup
sage: S = SymmetricGroup(3)
sage: gamma = S('(1,2)')
sage: G.relabel(gamma, inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

A way to get a random relabeling:

```
sage: set_random_seed(0)  # Results are reproducible
sage: D = DiGraph({1: [2], 3: [4]})
sage: D.relabel(Permutations(D.vertices()).random_element())
sage: D.sources()
[1, 4]
```

Relabeling using an injective function:

```
sage: G.edges()
[(0, 1, None), (1, 2, None)]
sage: H = G.relabel(lambda i: i+10, inplace=False)
sage: H.vertices()
[10, 11, 12]
sage: H.edges()
[(10, 11, None), (11, 12, None)]
```

Relabeling using a non injective function has no meaning:

```
sage: G.edges()
[(0, 1, None), (1, 2, None)]
sage: G.relabel(lambda i: 0, inplace=False)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: Non injective relabeling
```

But this test can be disabled, which can lead to ... problems:

```
sage: G.edges()
[(0, 1, None), (1, 2, None)]
sage: G.relabel(lambda i: 0, check_input = False)
sage: G.edges()
[]
```

Recovering the relabeling with `return_map`:

```
sage: G = graphs.CubeGraph(3)
sage: G.relabel(range(8), return_map=True)
{'000': 0,
 '001': 1,
 '010': 2,
 '011': 3,
 '100': 4,
 '101': 5,
 '110': 6,
 '111': 7}
```

When no permutation is given, the relabeling is done to integers from 0 to N-1 but in an arbitrary order:

```
sage: G = graphs.CubeGraph(3)
sage: G.vertices()
['000', '001', '010', '011', '100', '101', '110', '111']
sage: G.relabel()
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7]
```

In the above case, the mapping is arbitrary but consistent:

```
sage: map1 = G.relabel(inplace=False, return_map=True)
sage: map2 = G.relabel(inplace=False, return_map=True)
sage: map1 == map2
True
```

```
sage: G = graphs.PathGraph(3)
sage: G.relabel(lambda i: i+10, return_map=True)
{0: 10, 1: 11, 2: 12}
```

remove_loops(*vertices=None*)

Remove loops on vertices in `vertices`.

If `vertices` is `None`, removes all loops.

EXAMPLES:

```
sage: G = Graph(4, loops=True)
sage: G.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: G.remove_loops()
```

(continues on next page)

(continued from previous page)

```
sage: G.edges(labels=False)
[(2, 3)]
sage: G.allows_loops()
True
sage: G.has_loops()
False

sage: D = DiGraph(4, loops=True)
sage: D.add_edges([(0, 0), (1, 1), (2, 2), (3, 3), (2, 3)])
sage: D.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: D.remove_loops()
sage: D.edges(labels=False)
[(2, 3)]
sage: D.allows_loops()
True
sage: D.has_loops()
False
```

remove_multiple_edges()

Remove all multiple edges, retaining one edge for each.

See also:

See also *allow_multiple_edges()*

EXAMPLES:

```
sage: G = Graph(multiedges=True, sparse=True)
sage: G.add_edges([(0,1), (0,1), (0,1), (0,1), (1,2)])
sage: G.edges(labels=False)
[(0, 1), (0, 1), (0, 1), (0, 1), (1, 2)]
```

```
sage: G.remove_multiple_edges()
sage: G.edges(labels=False)
[(0, 1), (1, 2)]
```

```
sage: D = DiGraph(multiedges=True, sparse=True)
sage: D.add_edges([(0, 1, 1), (0, 1, 2), (0, 1, 3), (0, 1, 4), (1, 2, None)])
sage: D.edges(labels=False)
[(0, 1), (0, 1), (0, 1), (0, 1), (1, 2)]
sage: D.remove_multiple_edges()
sage: D.edges(labels=False)
[(0, 1), (1, 2)]
```

set_edge_label(*u*, *v*, *l*)

Set the edge label of a given edge.

Note: There can be only one edge from u to v for this to make sense. Otherwise, an error is raised.

INPUT:

- `u, v` – the vertices (and direction if digraph) of the edge
- `l` – the new label

EXAMPLES:

```
sage: SD = DiGraph({1:[18,2], 2:[5,3], 3:[4,6], 4:[7,2], 5:[4], 6:[13,12],␣
↪7:[18,8,10], 8:[6,9,10], 9:[6], 10:[11,13], 11:[12], 12:[13], 13:[17,14],␣
↪14:[16,15], 15:[2], 16:[13], 17:[15,13], 18:[13]}, sparse=True)
sage: SD.set_edge_label(1, 18, 'discrete')
sage: SD.set_edge_label(4, 7, 'discrete')
sage: SD.set_edge_label(2, 5, 'h = 0')
sage: SD.set_edge_label(7, 18, 'h = 0')
sage: SD.set_edge_label(7, 10, 'aut')
sage: SD.set_edge_label(8, 10, 'aut')
sage: SD.set_edge_label(8, 9, 'label')
sage: SD.set_edge_label(8, 6, 'no label')
sage: SD.set_edge_label(13, 17, 'k > h')
sage: SD.set_edge_label(13, 14, 'k = h')
sage: SD.set_edge_label(17, 15, 'v_k finite')
sage: SD.set_edge_label(14, 15, 'v_k m.c.r.')
sage: posn = {1:[ 3,-3],  2:[0,2],  3:[0, 13],  4:[3,9],  5:[3,3],  6:[16,␣
↪13], 7:[6,1],  8:[6,6],  9:[6,11], 10:[9,1], 11:[10,6], 12:[13,6], 13:[16,
↪2], 14:[10,-6], 15:[0,-10], 16:[14,-6], 17:[16,-10], 18:[6,-4]}
sage: SD.plot(pos=posn, vertex_size=400, vertex_colors={'#FFFFFF
↪':list(range(1,19))}, edge_labels=True).show() # long time
```

```
sage: G = graphs.HeawoodGraph()
sage: for u,v,l in G.edges(sort=False):
....:  G.set_edge_label(u, v, '(' + str(u) + ',' + str(v) + ')')
sage: G.edges()
    [(0, 1, '(0,1)'),
     (0, 5, '(0,5)'),
     (0, 13, '(0,13)'),
     ...
     (11, 12, '(11,12)'),
     (12, 13, '(12,13)')]
```

```
sage: g = Graph({0: [0, 1, 1, 2]}, loops=True, multiedges=True, sparse=True)
sage: g.set_edge_label(0, 0, 'test')
sage: g.edges()
[(0, 0, 'test'), (0, 1, None), (0, 1, None), (0, 2, None)]
sage: g.add_edge(0, 0, 'test2')
sage: g.set_edge_label(0,0,'test3')
Traceback (most recent call last):
...
RuntimeError: cannot set edge label, since there are multiple edges from 0 to␣
↪0
```

```
sage: dg = DiGraph({0: [1], 1: [0]}, sparse=True)
sage: dg.set_edge_label(0, 1, 5)
sage: dg.set_edge_label(1, 0, 9)
sage: dg.outgoing_edges(1)
[(1, 0, 9)]
sage: dg.incoming_edges(1)
[(0, 1, 5)]
sage: dg.outgoing_edges(0)
[(0, 1, 5)]
sage: dg.incoming_edges(0)
[(1, 0, 9)]
```

```
sage: G = Graph({0: {1: 1}}, sparse=True)
sage: G.num_edges()
1
sage: G.set_edge_label(0, 1, 1)
sage: G.num_edges()
1
```

set_embedding(*embedding*)

Set a combinatorial embedding dictionary to `_embedding` attribute.

Dictionary is organized with vertex labels as keys and a list of each vertex's neighbors in clockwise order.

Dictionary is error-checked for validity.

INPUT:

- `embedding` – a dictionary

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.set_embedding({0: [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2, 4],
↪ 4: [0, 9, 3], 5: [0, 8, 7], 6: [8, 1, 9], 7: [9, 2, 5], 8: [3, 5, 6], 9:␣
↪[4, 6, 7]})
sage: G.set_embedding({'s': [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2,␣
↪4], 4: [0, 9, 3], 5: [0, 8, 7], 6: [8, 1, 9], 7: [9, 2, 5], 8: [3, 5, 6],␣
↪9: [4, 6, 7]})
Traceback (most recent call last):
...
ValueError: vertices in ['s'] from the embedding do not belong to the graph
```

set_latex_options(***kwds*)

Set multiple options for rendering a graph with LaTeX.

INPUT:

- `kwds` – any number of option/value pairs to set many graph latex options at once (a variable number, in any order). Existing values are overwritten, new values are added. Existing values can be cleared by setting the value to `None`. Possible options are documented at *sage.graphs.graph_latex.GraphLatex.set_option()*.

This method is a convenience for setting the options of a graph directly on an instance of the graph. For a full explanation of how to use LaTeX to render graphs, see the introduction to the *graph_latex* module.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.set_latex_options(tkz_style='Welsh')
sage: opts = g.latex_options()
sage: opts.get_option('tkz_style')
'Welsh'
```

set_planar_positions(*test=False*, ***layout_options*)

Compute a planar layout for self using Schnyder's algorithm, and save it as default layout.

EXAMPLES:

```
sage: g = graphs.CycleGraph(7)
sage: g.set_planar_positions(test=True)
doctest:...: DeprecationWarning: This method is replaced by the method layout.
↪ Please use layout(layout="planar", save_pos=True) instead.
```

(continues on next page)

(continued from previous page)

```
See http://trac.sagemath.org/24494 for details.
True
```

This method is deprecated since Sage-4.4.1.alpha2. Please use instead:

sage: g.layout(layout = “planar”, save_pos = True) {0: [1, 4], 1: [5, 1], 2: [0, 5], 3: [1, 0], 4: [1, 2], 5: [2, 1], 6: [4, 1]}

set_pos (*pos*, *dim=2*)

Set the position dictionary.

The position dictionary specifies the coordinates of each vertex.

INPUT:

- `pos` – a position dictionary for the vertices of the (di)graph
- `dim` – integer (default: 2); the number of coordinates per vertex

EXAMPLES:

Note that *`set_pos()`* will allow you to do ridiculous things, which will not blow up until plotting:

```
sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: (..., ...),
 ...
 9: (..., ...)}
```

```
sage: G.set_pos('spam')
sage: P = G.plot()
Traceback (most recent call last):
...
TypeError: string indices must be integers...
```

set_vertex (*vertex*, *object*)

Associate an arbitrary object with a vertex.

INPUT:

- `vertex` – which vertex
- `object` – object to associate to vertex

EXAMPLES:

```
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertex(1, graphs.FlowerSnark())
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
sage: T.set_vertex(4, 'foo')
Traceback (most recent call last):
...
ValueError: vertex (4) not in the graph
```

set_vertices (*vertex_dict*)

Associate arbitrary objects with each vertex, via an association dictionary.

INPUT:

- vertex_dict – the association dictionary

EXAMPLES:

```
sage: d = {0: graphs.DodecahedralGraph(), 1: graphs.FlowerSnark(), 2: graphs.
↪MoebiusKantorGraph(), 3: graphs.PetersenGraph()}
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

shortest_path(*u*, *v*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return a list of vertices representing some shortest path from u to v.

If there is no path from u to v, the returned list is empty.

For more information and more examples, see *shortest_paths()* (the inputs are very similar).

INPUT:

- u, v – the start and the end vertices of the paths
- by_weight – boolean (default: False); if True, the edges in the graph are weighted, otherwise all edges have weight 1
- algorithm – string (default: None); one of the following algorithms:
  - 'BFS': performs a BFS from u. Does not work with edge weights.
  - 'BFS_Bid': performs a BFS from u and from v. Does not work with edge weights.
  - 'Dijkstra_NetworkX': the Dijkstra algorithm, implemented in NetworkX. Works only with positive weights.
  - 'Dijkstra_Bid_NetworkX': performs a Dijkstra visit from u and from v (NetworkX implementation). Works only with positive weights.
  - 'Dijkstra_Bid': a Cython implementation that performs a Dijkstra visit from u and from v. Works only with positive weights.
  - 'Bellman-Ford_Boost': the Bellman-Ford algorithm, implemented in Boost. Works also with negative weights, if there is no negative cycle.
  - None (default): Sage chooses the best algorithm: 'BFS_Bid' if by_weight is False, 'Dijkstra_Bid' otherwise.

  Note: If there are negative weights and algorithm is None, the result is not reliable. This occurs because, for performance reasons, we cannot check whether there are edges with negative weights before running the algorithm. If there are, the user should explicitly input algorithm='Bellman-Ford_Boost'.

- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path(4, 9)
[4, 17, 16, 12, 13, 9]
sage: D.shortest_path(4, 9, algorithm='BFS')
[4, 3, 2, 1, 8, 9]
sage: D.shortest_path(4, 8, algorithm='Dijkstra_NetworkX')
[4, 3, 2, 1, 8]
sage: D.shortest_path(4, 8, algorithm='Dijkstra_Bid_NetworkX')
[4, 3, 2, 1, 8]
sage: D.shortest_path(4, 9, algorithm='Dijkstra_Bid')
[4, 3, 19, 0, 10, 9]
sage: D.shortest_path(5, 5)
[5]
sage: D.delete_edges(D.edges_incident(13))
sage: D.shortest_path(13, 4)
[]
sage: G = Graph({0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2}},␣
↪sparse = True)
sage: G.plot(edge_labels=True).show() # long time
sage: G.shortest_path(0, 3)
[0, 4, 3]
sage: G.shortest_path(0, 3, by_weight=True)
[0, 1, 2, 3]
sage: G.shortest_path(0, 3, by_weight=True, algorithm='Dijkstra_NetworkX')
[0, 1, 2, 3]
sage: G.shortest_path(0, 3, by_weight=True, algorithm='Dijkstra_Bid_NetworkX')
[0, 1, 2, 3]
```

shortest_path_all_pairs(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return a shortest path between each pair of vertices.

INPUT:

- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'BFS'`: the computation is done through a BFS centered on each vertex successively. Works only if `by_weight==False`.
  - `'Floyd-Warshall-Cython'`: the Cython implementation of the Floyd-Warshall algorithm. Works only if `by_weight==False`.
  - `'Floyd-Warshall-Python'`: the Python implementation of the Floyd-Warshall algorithm. Works also with weighted graphs, even with negative weights (but no negative cycle is allowed).
  - `'Floyd-Warshall_Boost'`: the Boost implementation of the Floyd-Warshall algorithm. Works also with weighted graphs, even with negative weights (but no negative cycle is allowed).
  - `'Dijkstra_NetworkX'`: the Dijkstra algorithm, implemented in NetworkX. It works with weighted graphs, but no negative weight is allowed.
  - `'Dijkstra_Boost'`: the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - `'Johnson_Boost'`: the Johnson algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).

  - None (default): Sage chooses the best algorithm: 'BFS' if by_weight is False, 'Dijkstra_Boost' if all weights are positive, 'Floyd-Warshall_Boost' otherwise.

- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

OUTPUT:

A tuple (dist, pred). They are both dicts of dicts. The first indicates the length dist[u][v] of the shortest weighted path from u to v. The second is a compact representation of all the paths - it indicates the predecessor pred[u][v] of v in the shortest path from u to v.

Note: Only reachable vertices are present in the dictionaries.

Note: There is a Cython version of this method that is usually much faster for large graphs, as most of the time is actually spent building the final double dictionary. Everything on the subject is to be found in the *distances_all_pairs* module.

EXAMPLES:

Some standard examples (see *shortest_paths()* for more examples on how to use the input variables):

```
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} },␣
↪sparse=True)
sage: G.plot(edge_labels=True).show() # long time
sage: dist, pred = G.shortest_path_all_pairs(by_weight = True)
sage: dist
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 2}, 1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 3}, 2: {0:␣
↪2, 1: 1, 2: 0, 3: 1, 4: 3}, 3: {0: 3, 1: 2, 2: 1, 3: 0, 4: 2}, 4: {0: 2, 1:␣
↪3, 2: 3, 3: 2, 4: 0}}
sage: pred
{0: {0: None, 1: 0, 2: 1, 3: 2, 4: 0}, 1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},␣
↪2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3}, 3: {0: 1, 1: 2, 2: 3, 3: None, 4: 3},␣
↪4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}}
sage: pred[0]
{0: None, 1: 0, 2: 1, 3: 2, 4: 0}
sage: G = Graph( { 0: {1: {'weight':1}}, 1: {2: {'weight':1}}, 2: {3: {'weight
↪':1}}, 3: {4: {'weight':2}}, 4: {0: {'weight':2}} }, sparse=True)
sage: dist, pred = G.shortest_path_all_pairs(weight_function = lambda e:e[2][
↪'weight'])
sage: dist
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 2}, 1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 3}, 2: {0:␣
↪2, 1: 1, 2: 0, 3: 1, 4: 3}, 3: {0: 3, 1: 2, 2: 1, 3: 0, 4: 2}, 4: {0: 2, 1:␣
↪3, 2: 3, 3: 2, 4: 0}}
sage: pred
{0: {0: None, 1: 0, 2: 1, 3: 2, 4: 0}, 1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},␣
↪2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3}, 3: {0: 1, 1: 2, 2: 3, 3: None, 4: 3},␣
↪4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}}
```

So for example the shortest weighted path from 0 to 3 is obtained as follows. The predecessor of 3 is pred[0][3] == 2, the predecessor of 2 is pred[0][2] == 1, and the predecessor of 1 is

pred[0][1] == 0.

```
sage: G = Graph( { 0: {1:None}, 1: {2:None}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }
↪, sparse=True )
sage: G.shortest_path_all_pairs()
({0: {0: 0, 1: 1, 2: 2, 3: 2, 4: 1},
1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 2},
2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 2},
3: {0: 2, 1: 2, 2: 1, 3: 0, 4: 1},
4: {0: 1, 1: 2, 2: 2, 3: 1, 4: 0}},
{0: {0: None, 1: 0, 2: 1, 3: 4, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3},
3: {0: 4, 1: 2, 2: 3, 3: None, 4: 3},
4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}})
sage: G.shortest_path_all_pairs(weight_function=lambda e:(e[2] if e[2] is not␣
↪None else 1))
({0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 2},
1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 3},
2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 3},
3: {0: 3, 1: 2, 2: 1, 3: 0, 4: 2},
4: {0: 2, 1: 3, 2: 3, 3: 2, 4: 0}},
{0: {0: None, 1: 0, 2: 1, 3: 2, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3},
3: {0: 1, 1: 2, 2: 3, 3: None, 4: 3},
4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}})
```

Checking that distances are equal regardless of the algorithm used:

```
sage: g = graphs.Grid2dGraph(5,5)
sage: d1, _ = g.shortest_path_all_pairs(algorithm="BFS")
sage: d2, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall-Cython")
sage: d3, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall-Python")
sage: d4, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_NetworkX")
sage: d5, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_Boost")
sage: d6, _ = g.shortest_path_all_pairs(algorithm="Johnson_Boost")
sage: d7, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall_Boost")
sage: d1 == d2 == d3 == d4 == d5 == d6 == d7
True
```

Checking that distances are equal regardless of the algorithm used:

```
sage: g = digraphs.RandomDirectedGNM(6,12)
sage: d1, _ = g.shortest_path_all_pairs(algorithm="BFS")
sage: d2, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall-Cython")
sage: d3, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall-Python")
sage: d4, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_NetworkX")
sage: d5, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_Boost")
sage: d6, _ = g.shortest_path_all_pairs(algorithm="Johnson_Boost")
sage: d7, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall_Boost")
sage: d1 == d2 == d3 == d4 == d5 == d6 == d7
True
```

Checking that weighted distances are equal regardless of the algorithm used:

```
sage: g = graphs.CompleteGraph(5)
sage: import random
```

(continues on next page)

(continued from previous page)

```
sage: for v, w in g.edges(labels=False, sort=False):
....:     g.add_edge(v, w, random.uniform(1, 10))
sage: d1, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall-Python")
sage: d2, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_NetworkX")
sage: d3, _ = g.shortest_path_all_pairs(algorithm="Dijkstra_Boost")
sage: d4, _ = g.shortest_path_all_pairs(algorithm="Johnson_Boost")
sage: d5, _ = g.shortest_path_all_pairs(algorithm="Floyd-Warshall_Boost")
sage: d1 == d2 == d3 == d4 == d5
True
```

Checking a random path is valid:

```
sage: dist, path = g.shortest_path_all_pairs(algorithm="BFS")
sage: u,v = g.random_vertex(), g.random_vertex()
sage: p = [v]
sage: while p[0] is not None:
....:   p.insert(0,path[u][p[0]])
sage: len(p) == dist[u][v] + 2
True
```

Negative weights:

```
sage: g = DiGraph([(0,1,-2),(1,0,1)], weighted=True)
sage: g.shortest_path_all_pairs(by_weight=True)
Traceback (most recent call last):
...
ValueError: the graph contains a negative cycle
```

Unreachable vertices are not present in the dictionaries:

```
sage: g = DiGraph([(0,1,1),(1,2,2)])
sage: g.shortest_path_all_pairs(algorithm='BFS')
({0: {0: 0, 1: 1, 2: 2}, 1: {1: 0, 2: 1}, 2: {2: 0}},
 {0: {0: None, 1: 0, 2: 1}, 1: {1: None, 2: 1}, 2: {2: None}})
sage: g.shortest_path_all_pairs(algorithm='Dijkstra_NetworkX')
({0: {0: 0, 1: 1, 2: 2}, 1: {1: 0, 2: 1}, 2: {2: 0}},
 {0: {0: None, 1: 1, 2: 1}, 1: {1: None, 2: 2}, 2: {2: None}})
sage: g.shortest_path_all_pairs(algorithm='Dijkstra_Boost')
({0: {0: 0, 1: 1, 2: 2}, 1: {1: 0, 2: 1}, 2: {2: 0}},
 {0: {0: None, 1: 0, 2: 1}, 1: {1: None, 2: 1}, 2: {2: None}})
sage: g.shortest_path_all_pairs(algorithm='Floyd-Warshall-Python')
({0: {0: 0, 1: 1, 2: 2}, 1: {1: 0, 2: 1}, 2: {2: 0}},
 {0: {0: None, 1: 0, 2: 1}, 1: {1: None, 2: 1}, 2: {2: None}})
sage: g.shortest_path_all_pairs(algorithm='Floyd-Warshall-Cython')
({0: {0: 0, 1: 1, 2: 2}, 1: {1: 0, 2: 1}, 2: {2: 0}},
 {0: {0: None, 1: 0, 2: 1}, 1: {1: None, 2: 1}, 2: {2: None}})
```

In order to change the default behavior if the graph is disconnected, we can use default values with dictionaries:

```
sage: G = 2*graphs.PathGraph(2)
sage: d,_ = G.shortest_path_all_pairs()
sage: import itertools
sage: from sage.rings.infinity import Infinity
sage: for u,v in itertools.combinations(G.vertex_iterator(), 2):
....:     print("dist({}, {}) = {}".format(u,v, d[u].get(v,+Infinity)))
```

(continues on next page)

(continued from previous page)

```
dist(0, 1) = 1
dist(0, 2) = +Infinity
dist(0, 3) = +Infinity
dist(1, 2) = +Infinity
dist(1, 3) = +Infinity
dist(2, 3) = 1
```

shortest_path_length(*u*, *v*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the minimal length of a path from `u` to `v`.

If there is no path from u to v, returns `Infinity`.

For more information and more examples, we refer to *shortest_path()* and *shortest_paths()*, which have very similar inputs.

INPUT:

- `u`, `v` – the start and the end vertices of the paths
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'BFS'`: performs a BFS from `u`. Does not work with edge weights.
  - `'BFS_Bid'`: performs a BFS from `u` and from `v`. Does not work with edge weights.
  - `'Dijkstra_NetworkX'`: the Dijkstra algorithm, implemented in NetworkX. Works only with positive weights.
  - `'Dijkstra_Bid_NetworkX'`: performs a Dijkstra visit from `u` and from `v` (NetworkX implementation). Works only with positive weights.
  - `'Dijkstra_Bid'`: a Cython implementation that performs a Dijkstra visit from `u` and from `v`. Works only with positive weights.
  - `'Bellman-Ford_Boost'`: the Bellman-Ford algorithm, implemented in Boost. Works also with negative weights, if there is no negative cycle.
  - `None` (default): Sage chooses the best algorithm: `'BFS_Bid'` if `by_weight` is `False`, `'Dijkstra_Bid'` otherwise.

  Note: If there are negative weights and algorithm is `None`, the result is not reliable. This occurs because, for performance reasons, we cannot check whether there are edges with negative weights before running the algorithm. If there are, the user should explicitly input `algorithm='Bellman-Ford_Boost'`.

- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); if `True`, we check that the weight_function outputs a number for each edge

EXAMPLES:

Standard examples:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path_length(4, 9)
5
sage: D.shortest_path_length(4, 9, algorithm='BFS')
5
sage: D.shortest_path_length(4, 9, algorithm='Dijkstra_NetworkX')
5
sage: D.shortest_path_length(4, 9, algorithm='Dijkstra_Bid_NetworkX')
5
sage: D.shortest_path_length(4, 9, algorithm='Dijkstra_Bid')
5
sage: D.shortest_path_length(4, 9, algorithm='Bellman-Ford_Boost')
5
sage: D.shortest_path_length(5, 5)
0
sage: D.delete_edges(D.edges_incident(13))
sage: D.shortest_path_length(13, 4)
+Infinity
sage: G = Graph({0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2}},␣
↪sparse = True)
sage: G.plot(edge_labels=True).show() # long time
sage: G.shortest_path_length(0, 3)
2
sage: G.shortest_path_length(0, 3, by_weight=True)
3
sage: G.shortest_path_length(0, 3, by_weight=True, algorithm='Dijkstra_
↪NetworkX')
3
sage: G.shortest_path_length(0, 3, by_weight=True, algorithm='Dijkstra_Bid_
↪NetworkX')
3
```

If Dijkstra is used with negative weights, usually it raises an error:

```
sage: G = DiGraph({0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: -2}},␣
↪sparse = True)
sage: G.shortest_path_length(4, 1, by_weight=True, algorithm=None)
Traceback (most recent call last):
...
ValueError: the graph contains an edge with negative weight
sage: G.shortest_path_length(4, 1, by_weight=True, algorithm='Bellman-Ford_
↪Boost')
-1
```

However, sometimes the result may be wrong, and no error is raised:

```
sage: G = DiGraph([(0,1,1),(1,2,1),(0,3,1000),(3,4,-3000), (4,2,1000)])
sage: G.shortest_path_length(0, 2, by_weight=True, algorithm='Bellman-Ford_
↪Boost')
-1000
sage: G.shortest_path_length(0, 2, by_weight=True)
2
```

shortest_path_lengths(*u*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the length of a shortest path from `u` to any other vertex.

Returns a dictionary of shortest path lengths keyed by targets, excluding all vertices that are not reachable

from u.

For more information on the input variables and more examples, we refer to *shortest_paths()* which has the same input variables.

INPUT:

- u – the starting vertex
- by_weight – boolean (default: False); if True, the edges in the graph are weighted, otherwise all edges have weight 1
- algorithm – string (default: None); one of the following algorithms:
  - 'BFS': performs a BFS from u. Does not work with edge weights.
  - 'Dijkstra_NetworkX': the Dijkstra algorithm, implemented in NetworkX (works only with positive weights).
  - 'Dijkstra_Boost': the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - 'Bellman-Ford_Boost': the Bellman-Ford algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - None (default): Sage chooses the best algorithm: 'BFS' if by_weight is False, 'Dijkstra_Boost' if all weights are positive, 'Bellman-Ford_Boost' otherwise.
- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

Unweighted case:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path_lengths(0)
{0: 0, 1: 1, 2: 2, 3: 2, 4: 3, 5: 4, 6: 3, 7: 3, 8: 2, 9: 2, 10: 1, 11: 2,␣
↪12: 3, 13: 3, 14: 4, 15: 5, 16: 4, 17: 3, 18: 2, 19: 1}
```

Weighted case:

```
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} },␣
↪sparse=True)
sage: G.plot(edge_labels=True).show() # long time
sage: G.shortest_path_lengths(0, by_weight=True)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 2}
```

Using a weight function:

```
sage: D = DiGraph([(0,1,{'weight':1}),(1,2,{'weight':3}),(0,2,{'weight':5})])
sage: weight_function = lambda e:e[2]['weight']
sage: D.shortest_path_lengths(1, algorithm='Dijkstra_NetworkX', by_
↪weight=False)
{1: 0, 2: 1}
sage: D.shortest_path_lengths(0, weight_function=weight_function)
{0: 0, 1: 1, 2: 4}
sage: D.shortest_path_lengths(1, weight_function=weight_function)
{1: 0, 2: 3}
```

Negative weights:

```
sage: D = DiGraph([(0,1,{'weight':-1}),(1,2,{'weight':3}),(0,2,{'weight':5})])
sage: D.shortest_path_lengths(0, weight_function=weight_function)
{0: 0, 1: -1, 2: 2}
```

Negative cycles:

```
sage: D = DiGraph([(0,1,{'weight':-5}),(1,2,{'weight':3}),(2,0,{'weight':1})])
sage: D.shortest_path_lengths(0, weight_function=weight_function)
Traceback (most recent call last):
...
ValueError: the graph contains a negative cycle
```

Checking that distances are equal regardless of the algorithm used:

```
sage: g = graphs.Grid2dGraph(5,5)
sage: d1 = g.shortest_path_lengths((0,0), algorithm="BFS")
sage: d2 = g.shortest_path_lengths((0,0), algorithm="Dijkstra_NetworkX")
sage: d3 = g.shortest_path_lengths((0,0), algorithm="Dijkstra_Boost")
sage: d4 = g.shortest_path_lengths((0,0), algorithm="Bellman-Ford_Boost")
sage: d1 == d2 == d3 == d4
True
```

shortest_paths(*u*, *by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*, *cutoff=None*)

Return a dictionary associating to each vertex `v` a shortest path from `u` to `v`, if it exists.

If u and v are not connected, vertex v is not present in the dictionary.

INPUT:

- `u` – the starting vertex
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'BFS'`: performs a BFS from `u`. Does not work with edge weights.
  - `'Dijkstra_NetworkX'`: the Dijkstra algorithm, implemented in NetworkX (works only with positive weights).
  - `'Dijkstra_Boost'`: the Dijkstra algorithm, implemented in Boost (works only with positive weights).
  - `'Bellman-Ford_Boost'`: the Bellman-Ford algorithm, implemented in Boost (works also with negative weights, if there is no negative cycle).
  - `None` (default): Sage chooses the best algorithm: `'BFS'` if `by_weight` is `False`, `'Dijkstra_Boost'` if all weights are positive, `'Bellman-Ford_Boost'` otherwise.
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `check_weight` – boolean (default: `True`); if `True`, we check that the weight_function outputs a number for each edge
- `cutoff` – integer (default: `None`); integer depth to stop search (used only if `algorithm=='BFS'`)

EXAMPLES:

Standard example:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_paths(0)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 19, 3], 4: [0, 19, 3, 4], 5: [0, 1,␣
↪2, 6, 5], 6: [0, 1, 2, 6], 7: [0, 1, 8, 7], 8: [0, 1, 8], 9: [0, 10, 9],␣
↪10: [0, 10], 11: [0, 10, 11], 12: [0, 10, 11, 12], 13: [0, 10, 9, 13], 14:␣
↪[0, 1, 8, 7, 14], 15: [0, 19, 18, 17, 16, 15], 16: [0, 19, 18, 17, 16], 17:␣
↪[0, 19, 18, 17], 18: [0, 19, 18], 19: [0, 19]}
```

All these paths are obviously induced graphs:

```
sage: all(D.subgraph(p).is_isomorphic(graphs.PathGraph(len(p))) for p in D.
↪shortest_paths(0).values())
True
```

```
sage: D.shortest_paths(0, cutoff=2)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 19, 3], 8: [0, 1, 8], 9: [0, 10, 9],␣
↪10: [0, 10], 11: [0, 10, 11], 18: [0, 19, 18], 19: [0, 19]}
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} },␣
↪sparse=True)
sage: G.plot(edge_labels=True).show() # long time
sage: G.shortest_paths(0, by_weight=True)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 4]}
```

Weighted shortest paths:

```
sage: D = DiGraph([(0,1,1),(1,2,3),(0,2,5)])
sage: D.shortest_paths(0)
{0: [0], 1: [0, 1], 2: [0, 2]}
sage: D.shortest_paths(0, by_weight=True)
{0: [0], 1: [0, 1], 2: [0, 1, 2]}
```

Using a weight function (this way, `by_weight` is set to `True`):

```
sage: D = DiGraph([(0,1,{'weight':1}),(1,2,{'weight':3}),(0,2,{'weight':5})])
sage: weight_function = lambda e:e[2]['weight']
sage: D.shortest_paths(0, weight_function=weight_function)
{0: [0], 1: [0, 1], 2: [0, 1, 2]}
```

If the weight function does not match the label:

```
sage: D.shortest_paths(0, weight_function=lambda e:e[2])
Traceback (most recent call last):
...
ValueError: the weight function cannot find the weight of (0, 1, {'weight': 1}
↪)
```

However, if `check_weight` is set to `False`, unexpected behavior may occur:

```
sage: D.shortest_paths(0, algorithm='Dijkstra_NetworkX', weight_
↪function=lambda e:e[2], check_weight=False)
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'int' and 'dict'
```

Negative weights:

```
sage: D = DiGraph([(0,1,1),(1,2,-2),(0,2,4)])
sage: D.shortest_paths(0, by_weight=True)
{0: [0], 1: [0, 1], 2: [0, 1, 2]}
```

Negative cycles:

```
sage: D.add_edge(2,0,0)
sage: D.shortest_paths(0, by_weight=True)
Traceback (most recent call last):
...
ValueError: the graph contains a negative cycle
```

shortest_simple_paths(*source*, *target*, *weight_function=None*, *by_weight=False*, *algorithm=None*, *report_edges=False*, *labels=False*, *report_weight=False*)

Return an iterator over the simple paths between a pair of vertices.

This method returns an iterator over the simple paths (i.e., without repetition) from `source` to `target`. By default (`by_weight` is `False`), the paths are reported by increasing number of edges. When `by_weight` is `True`, the paths are reported by increasing weights.

In case of weighted graphs negative weights are not allowed.

If `source` is the same vertex as `target`, then `[[source]]` is returned – a list containing the 1-vertex, 0-edge path `source`.

By default `Yen's` algorithm [Yen1970] is used for undirected graphs and `Feng's` algorithm is used for directed graphs [Feng2014].

The loops and the multiedges if present in the given graph are ignored and only minimum of the edge labels is kept in case of multiedges.

INPUT:

- `source` – a vertex of the graph, where to start
- `target` – a vertex of the graph, where to end
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); the algorithm to use in computing `k` shortest paths of `self`. The following algorithms are supported:
  - `"Yen"` – Yen's algorithm [Yen1970]
  - `"Feng"` – an improved version of Yen's algorithm but that works only for directed graphs [Feng2014]
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges. When set to `False`, the `labels` parameter is ignored.
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.
- `report_weight` – boolean (default: `False`); if `False`, just the path between `source` and `target` is returned. Otherwise a tuple of path length and path is returned.

EXAMPLES:

```
sage: g = DiGraph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10),
↪ (4, 5, 30)])
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Yen"))
[[1, 3, 5], [1, 2, 5], [1, 4, 5]]
sage: list(g.shortest_simple_paths(1, 5, algorithm="Yen"))
[[1, 2, 5], [1, 3, 5], [1, 4, 5]]
sage: list(g.shortest_simple_paths(1, 1))
[[1]]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, report_edges=True,␣
↪report_weight=True, labels=True))
[(20, [(1, 3, 10), (3, 5, 10)]),
 (40, [(1, 2, 20), (2, 5, 20)]),
 (60, [(1, 4, 30), (4, 5, 30)])]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Feng",␣
↪report_edges=True, report_weight=True))
[(20, [(1, 3), (3, 5)]), (40, [(1, 2), (2, 5)]), (60, [(1, 4), (4, 5)])]
sage: list(g.shortest_simple_paths(1, 5, report_edges=True, report_
↪weight=True))
[(2, [(1, 4), (4, 5)]), (2, [(1, 3), (3, 5)]), (2, [(1, 2), (2, 5)])]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, report_edges=True))
[[(1, 3), (3, 5)], [(1, 2), (2, 5)], [(1, 4), (4, 5)]]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Feng",␣
↪report_edges=True, labels=True))
[[(1, 3, 10), (3, 5, 10)], [(1, 2, 20), (2, 5, 20)], [(1, 4, 30), (4, 5, 30)]]
sage: g = Graph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10),␣
↪(4, 5, 30), (1, 6, 100), (5, 6, 5)])
sage: list(g.shortest_simple_paths(1, 6, by_weight = True))
[[1, 3, 5, 6], [1, 2, 5, 6], [1, 4, 5, 6], [1, 6]]
sage: list(g.shortest_simple_paths(1, 6, algorithm="Yen"))
[[1, 6], [1, 2, 5, 6], [1, 3, 5, 6], [1, 4, 5, 6]]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, report_
↪weight=True, labels=True))
[(1, [(1, 6, 100)]),
 (3, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (3, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (3, [(1, 4, 30), (4, 5, 30), (5, 6, 5)])]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, report_
↪weight=True, labels=True, by_weight=True))
[(25, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (45, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (65, [(1, 4, 30), (4, 5, 30), (5, 6, 5)]),
 (100, [(1, 6, 100)])]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, labels=True, by_
↪weight=True))
[[(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)],
 [(1, 6, 100)]]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, labels=True))
[[(1, 6, 100)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)]]
```

show(*method='matplotlib'*, ***kwds*)

Show the (di)graph.

INPUT:

- `method` – string (default: `"matplotlib"`); method to use to display the graph, either `"matplotlib"`, or `"js"` to visualize it in a browser using d3.js.
- Any other argument supported by the drawing functions:
  - `"matplotlib"` – see *GenericGraph.plot* and `sage.plot.graphics.Graphics.show()`
  - `"js"` – see *gen_html_code()*

EXAMPLES:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.plot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()  # long time (3s on sage.math, 2011)
```

show3d(*bgcolor=(1, 1, 1)*, *vertex_colors=None*, *vertex_size=0.06*, *edge_colors=None*, *edge_size=0.02*, *edge_size2=0.0325*, *pos3d=None*, *color_by_label=False*, *engine='threejs'*, ***kwds*)

Plot the graph and show the resulting plot.

INPUT:

- `bgcolor` – rgb tuple (default: `(1,1,1)`)
- `vertex_size` – float (default: `0.06`)
- `vertex_labels` – a boolean (default: `False`); whether to display vertices using text labels instead of spheres
- `vertex_colors` – dictionary (default: `None`); optional dictionary to specify vertex colors: each key is a color recognizable by `tachyon` (rgb tuple (default: `(1,0,0)`)), and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it doesn't get drawn).
- `edge_colors` – dictionary (default: `None`); a dictionary specifying edge colors: each key is a color recognized by `tachyon` (default: `(0,0,0)`), and each entry is a list of edges.
- `color_by_label` – a boolean or dictionary or function (default: `False`) whether to color each edge with a different color according to its label; the colors are chosen along a rainbow, unless they are specified by a function or dictionary mapping labels to colors; this option is incompatible with `edge_color` and `edge_colors`.
- `edge_size` – float (default: `0.02`)
- `edge_size2` – float (default: `0.0325`); used for `Tachyon` sleeves
- `pos3d` – a position dictionary for the vertices
- `layout`, `iterations`, ... – layout options; see *layout()*
- `engine` – string (default: `'threejs'`); the renderer to use among:
  - `'threejs'`: interactive web-based 3D viewer using JavaScript and a WebGL renderer
  - `'jmol'`: interactive 3D viewer using Java
  - `'tachyon'`: ray tracer generating a static PNG image
- `xres` – resolution
- `yres` – resolution
- `**kwds` – passed on to the rendering engine

EXAMPLES:

```
sage: G = graphs.CubeGraph(5)
sage: G.show3d(iterations=500, edge_size=None, vertex_size=0.04) # long time
```

We plot a fairly complicated Cayley graph:

```
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.show3d(vertex_size=0.03, edge_size=0.01, edge_size2=0.02, vertex_
↪colors={(1,1,1): list(G)}, bgcolor=(0,0,0), color_by_label=True,␣
↪iterations=200) # long time
```

Some `Tachyon` examples:

```
sage: D = graphs.DodecahedralGraph()
sage: D.show3d(engine='tachyon') # long time
```

```
sage: G = graphs.PetersenGraph()
sage: G.show3d(engine='tachyon', vertex_colors={(0,0,1): list(G)}) # long time
```

```
sage: C = graphs.CubeGraph(4)
sage: C.show3d(engine='tachyon', edge_colors={(0,1,0): C.edges(sort=False)},␣
↪vertex_colors={(1,1,1): list(C)}, bgcolor=(0,0,0)) # long time
```

```
sage: K = graphs.CompleteGraph(3)
sage: K.show3d(engine='tachyon', edge_colors={(1,0,0): [(0, 1, None)], (0, 1,␣
↪0): [(0, 2, None)], (0, 0, 1): [(1, 2, None)]}) # long time
```

size()

Return the number of edges.

Note that *num_edges()* also returns the number of edges in G.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

spanning_trees_count(*root_vertex=None*)

Return the number of spanning trees in a graph.

In the case of a digraph, counts the number of spanning out-trees rooted in `root_vertex`. Default is to set first vertex as root.

This computation uses Kirchhoff's Matrix Tree Theorem [1] to calculate the number of spanning trees. For complete graphs on n vertices the result can also be reached using Cayley's formula: the number of spanning trees are $n^(n-2)$.

For digraphs, the augmented Kirchhoff Matrix as defined in [2] is used for calculations. Here the result is the number of out-trees rooted at a specific vertex.

INPUT:

- `root_vertex` – a vertex (default: `None`); the vertex that will be used as root for all spanning out-trees if the graph is a directed graph. Otherwise, the first vertex returned by *vertex_iterator()* is used. This argument is ignored if the graph is not a digraph.

See also:

spanning_trees() – enumerates all spanning trees of a graph

REFERENCES:

- [1] http://mathworld.wolfram.com/MatrixTreeTheorem.html
- [2] Lih-Hsing Hsu, Cheng-Kuan Lin, "Graph Theory and Interconnection Networks"

AUTHORS:

- Anders Jonsson (2009-10-10)

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.spanning_trees_count()
2000
```

```
sage: n = 11
sage: G = graphs.CompleteGraph(n)
sage: ST = G.spanning_trees_count()
sage: ST == n ^ (n - 2)
True
```

```
sage: M = matrix(3, 3, [0, 1, 0, 0, 0, 1, 1, 1, 0])
sage: D = DiGraph(M)
sage: D.spanning_trees_count()
1
sage: D.spanning_trees_count(0)
1
sage: D.spanning_trees_count(2)
2
```

spectral_radius(*G*, *prec=1e-10*)

Return an interval of floating point number that encloses the spectral radius of this graph

The input graph `G` must be *strongly connected*.

INPUT:

- `prec` – (default `1e-10`) an upper bound for the relative precision of the interval

The algorithm is iterative and uses an inequality valid for non-negative matrices. Namely, if A is a non-negative square matrix with Perron-Frobenius eigenvalue λ then the following inequality is valid for any vector x

$$\min_i \frac{(Ax)_i}{x_i} \leq \lambda \leq \max_i \frac{(Ax)_i}{x_i}$$

Note: The speed of convergence of the algorithm is governed by the spectral gap (the distance to the second largest modulus of other eigenvalues). If this gap is small, then this function might not be appropriate.

The algorithm is not smart and not parallel! It uses basic interval arithmetic and native floating point arithmetic.

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import spectral_radius

sage: G = DiGraph([(0,0),(0,1),(1,0)], loops=True)
sage: phi = (RR(1) + RR(5).sqrt() ) / 2
sage: phi  # abs tol 1e-14
1.618033988749895
sage: e_min, e_max = spectral_radius(G, 1e-14)
sage: e_min, e_max     # abs tol 1e-14
(1.618033988749894, 1.618033988749896)
sage: (e_max - e_min)  # abs tol 1e-14
1e-14
sage: e_min < phi < e_max
True
```

This function also works for graphs:

```
sage: G = Graph([(0,1),(0,2),(1,2),(1,3),(2,4),(3,4)])
sage: e_min, e_max = spectral_radius(G, 1e-14)
sage: e = max(G.adjacency_matrix().charpoly().roots(AA, multiplicities=False))
sage: e_min < e < e_max
True

sage: G.spectral_radius()  # abs tol 1e-9
(2.48119430408, 2.4811943041)
```

A larger example:

```
sage: G = DiGraph()
sage: G.add_edges((i,i+1) for i in range(200))
sage: G.add_edge(200,0)
sage: G.add_edge(1,0)
sage: e_min, e_max = spectral_radius(G, 0.00001)
sage: p = G.adjacency_matrix(sparse=True).charpoly()
sage: p
x^201 - x^199 - 1
sage: r = p.roots(AA, multiplicities=False)[0]
sage: e_min < r < e_max
True
```

A much larger example:

```
sage: G = DiGraph(100000)
sage: r = list(range(100000))
sage: while not G.is_strongly_connected():
....:     shuffle(r)
....:     G.add_edges(enumerate(r), loops=False)
sage: spectral_radius(G, 1e-10)  # random
(1.9997956006500042, 1.9998043797692782)
```

The algorithm takes care of multiple edges:

```
sage: G = DiGraph(2,loops=True,multiedges=True)
sage: G.add_edges([(0,0),(0,0),(0,1),(1,0)])
sage: spectral_radius(G, 1e-14)  # abs tol 1e-14
(2.414213562373094, 2.414213562373095)
sage: max(G.adjacency_matrix().eigenvalues(AA))
2.414213562373095?
```

Some bipartite graphs:

```
sage: G = Graph([(0,1),(0,3),(2,3)])
sage: G.spectral_radius()  # abs tol 1e-10
(1.6180339887253428, 1.6180339887592732)

sage: G = DiGraph([(0,1),(0,3),(2,3),(3,0),(1,0),(1,2)])
sage: G.spectral_radius() # abs tol 1e-10
(1.5537739740270458, 1.553773974033029)

sage: G = graphs.CompleteBipartiteGraph(1,3)
sage: G.spectral_radius()  # abs tol 1e-10
(1.7320508075688772, 1.7320508075688774)
```

spectrum(*laplacian=False*)

Return a list of the eigenvalues of the adjacency matrix.

INPUT:

- `laplacian` – boolean (default: `False`); if `True`, use the Laplacian matrix (see *kirchhoff_matrix()*)

OUTPUT:

A list of the eigenvalues, including multiplicities, sorted with the largest eigenvalue first.

See also:

The method *spectral_radius()* returns floating point approximation of the maximum eigenvalue.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.spectrum()
[3, 1, 1, 1, 1, 1, -2, -2, -2, -2]
sage: P.spectrum(laplacian=True)
[5, 5, 5, 5, 2, 2, 2, 2, 2, 0]
sage: D = P.to_directed()
sage: D.delete_edge(7, 9)
sage: D.spectrum()
[2.9032119259..., 1, 1, 1, 1, 0.8060634335..., -1.7092753594..., -2, -2, -2]
```

```
sage: C = graphs.CycleGraph(8)
sage: C.spectrum()
[2, 1.4142135623..., 1.4142135623..., 0, 0, -1.4142135623..., -1.4142135623...
↪, -2]
```

A digraph may have complex eigenvalues. Previously, the complex parts of graph eigenvalues were being dropped. For a 3-cycle, we have:

```
sage: T = DiGraph({0: [1], 1: [2], 2: [0]})
sage: T.spectrum()
[1, -0.5000000000... + 0.8660254037...*I, -0.5000000000... - 0.8660254037...
↪*I]
```

steiner_tree(*vertices*, *weighted=False*, *solver=None*, *verbose=0*)

Return a tree of minimum weight connecting the given set of vertices.

Definition :

Computing a minimum spanning tree in a graph can be done in $n \log(n)$ time (and in linear time if all weights are equal) where $n = V + E$. On the other hand, if one is given a large (possibly weighted) graph and a subset of its vertices, it is NP-Hard to find a tree of minimum weight connecting the given set of vertices, which is then called a Steiner Tree.

See the Wikipedia article Steiner_tree_problem for more information.

INPUT:

- `vertices` – the vertices to be connected by the Steiner Tree.
- `weighted` – boolean (default: `False`); whether to consider the graph as weighted, and use each edge's label as a weight, considering `None` as a weight of 1. If `weighted=False` (default) all edges are considered to have a weight of 1.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`), sets the level of verbosity. Set to 0 by default, which means quiet.

Note:

- This problem being defined on undirected graphs, the orientation is not considered if the current graph is actually a digraph.
- The graph is assumed not to have multiple edges.

ALGORITHM:

Solved through Linear Programming.

COMPLEXITY:

NP-Hard.

Note that this algorithm first checks whether the given set of vertices induces a connected graph, returning one of its spanning trees if `weighted` is set to `False`, and thus answering very quickly in some cases

EXAMPLES:

The Steiner Tree of the first 5 vertices in a random graph is, of course, always a tree:

```
sage: g = graphs.RandomGNP(30, .5)
sage: first5 = g.vertices()[:5]
sage: st = g.steiner_tree(first5)
sage: st.is_tree()
True
```

And all the 5 vertices are contained in this tree

```
sage: all(v in st for v in first5)
True
```

An exception is raised when the problem is impossible, i.e. if the given vertices are not all included in the same connected component:

```
sage: g = 2 * graphs.PetersenGraph()
sage: st = g.steiner_tree([5, 15])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
EmptySetError: the given vertices do not all belong to the same connected␣
↪component. This problem has no solution !
```

strong_product (*other*)

Return the strong product of `self` and `other`.

The strong product of G and H is the graph L with vertex set $V(L) = V(G) \times V(H)$, and $((u, v), (w, x))$ is an edge of L iff either :

- (u, w) is an edge of G and $v = x$, or
- (v, x) is an edge of H and $u = w$, or
- (u, w) is an edge of G and (v, x) is an edge of H.

In other words, the edges of the strong product is the union of the edges of the tensor and Cartesian products.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: S = C.strong_product(Z); S
Graph on 10 vertices
sage: S.plot() # long time
Graphics object consisting of 36 graphics primitives
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: S = D.strong_product(P); S
Graph on 200 vertices
sage: S.plot() # long time
Graphics object consisting of 1701 graphics primitives
```

subdivide_edge (**args*)

Subdivide an edge k times.

INPUT:

The following forms are all accepted to subdivide 8 times the edge between vertices 1 and 2 labeled with `"my_label"`.

- `G.subdivide_edge( 1, 2, 8 )`
- `G.subdivide_edge( (1, 2), 8 )`
- `G.subdivide_edge( (1, 2, "my_label"), 8 )`

Note:

- If the given edge is labelled with l, all the edges created by the subdivision will have the same label
- If no label is given, the label used will be the one returned by the method *edge_label()* on the pair `u,v`

EXAMPLES:

Subdividing 5 times an edge in a path of length 3 makes it a path of length 8:

```
sage: g = graphs.PathGraph(3)
sage: edge = next(g.edge_iterator())
sage: g.subdivide_edge(edge, 5)
sage: g.is_isomorphic(graphs.PathGraph(8))
True
```

Subdividing a labelled edge in two ways:

```
sage: g = Graph()
sage: g.add_edge(0, 1, "label1")
sage: g.add_edge(1, 2, "label2")
sage: print(g.edges())
[(0, 1, 'label1'), (1, 2, 'label2')]
```

Specifying the label:

```
sage: g.subdivide_edge(0, 1, "label1", 3)
sage: print(g.edges())
[(0, 3, 'label1'), (1, 2, 'label2'), (1, 5, 'label1'), (3, 4, 'label1'), (4,␣
↪5, 'label1')]
```

The lazy way:

```
sage: g.subdivide_edge(1, 2, "label2", 5)
sage: print(g.edges())
[(0, 3, 'label1'), (1, 5, 'label1'), (1, 6, 'label2'), (2, 10, 'label2'), (3,␣
↪4, 'label1'), (4, 5, 'label1'), (6, 7, 'label2'), (7, 8, 'label2'), (8, 9,
↪'label2'), (9, 10, 'label2')]
```

If too many arguments are given, an exception is raised

```
sage: g.subdivide_edge(0,1,1,1,1,1,1,1,1,1,1)
Traceback (most recent call last):
...
ValueError: this method takes at most 4 arguments
```

The same goes when the given edge does not exist:

```
sage: g.subdivide_edge(0, 1, "fake_label", 5)
Traceback (most recent call last):
...
ValueError: the given edge does not exist
```

See also:

- *subdivide_edges()* – subdivides multiples edges at a time

subdivide_edges(*edges*, *k*)

Subdivide k times edges from an iterable container.

For more information on the behaviour of this method, please refer to the documentation of *subdivide_edge()*.

INPUT:

- `edges` – a list of edges
- `k` – integer; common length of the subdivisions

Note: If a given edge is labelled with l, all the edges created by its subdivision will have the same label.

EXAMPLES:

If we are given the disjoint union of several paths:

```
sage: paths = [2, 5, 9]
sage: paths = map(graphs.PathGraph, paths)
sage: g = Graph()
sage: for P in paths:
....:   g = g + P
```

Subdividing edges in each of them will only change their lengths:

```
sage: edges = [next(P.edge_iterator()) for P in g.connected_components_
↪subgraphs()]
sage: k = 6
sage: g.subdivide_edges(edges, k)
```

Let us check this by creating the graph we expect to have built through subdivision:

```
sage: paths2 = [2 + k, 5 + k, 9 + k]
sage: paths2 = map(graphs.PathGraph, paths2)
sage: g2 = Graph()
sage: for P in paths2:
....:   g2 = g2 + P
sage: g.is_isomorphic(g2)
True
```

See also:

- *subdivide_edge()* – subdivides one edge

subgraph(*vertices=None*, *edges=None*, *inplace=False*, *vertex_property=None*, *edge_property=None*, *algorithm=None*, *immutable=None*)

Return the subgraph containing the given vertices and edges.

If either vertices or edges are not specified, they are assumed to be all vertices or edges. If edges are not specified, returns the subgraph induced by the vertices.

INPUT:

- `inplace` – boolean (default: `False`); using `inplace=True` will simply delete the extra vertices and edges from the current graph. This will modify the graph.
- `vertices` – a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed (i.e., `None`), defaults to the entire graph.
- `edges` – as with `vertices`, edges can be a single edge or an iterable container of edges (e.g., a list, set, file, numeric array, etc.). By default (`edges=None`), all edges are assumed and the returned graph is an induced subgraph. In the case of multiple edges, specifying an edge as (u, v) means to keep all edges (u, v), regardless of the label.
- `vertex_property` – function (default: `None`); a function that inputs a vertex and outputs a boolean value, i.e., a vertex `v` in `vertices` is kept if `vertex_property(v) == True`
- `edge_property` – function (default: `None`); a function that inputs an edge and outputs a boolean value, i.e., a edge `e` in `edges` is kept if `edge_property(e) == True`

- `algorithm` – string (default: `None`); one of the following:
  - If `algorithm="delete"` or `inplace=True`, then the graph is constructed by deleting edges and vertices
  - If `algorithm="add"`, then the graph is constructed by building a new graph from the appropriate vertices and edges. Implies `inplace=False`.
  - If `algorithm=None`, then the algorithm is chosen based on the number of vertices in the subgraph.
- `immutable` – boolean (default: `None`); whether to create a mutable/immutable subgraph. `immutable=None` (default) means that the graph and its subgraph will behave the same way.

EXAMPLES:

```
sage: G = graphs.CompleteGraph(9)
sage: H = G.subgraph([0, 1, 2]); H
Subgraph of (Complete graph): Graph on 3 vertices
sage: G
Complete graph: Graph on 9 vertices
sage: J = G.subgraph(edges=[(0, 1)])
sage: J.edges(labels=False)
[(0, 1)]
sage: J.vertices() == G.vertices()
True
sage: G.subgraph([0, 1, 2], inplace=True); G
Subgraph of (Complete graph): Graph on 3 vertices
sage: G.subgraph() == G
True
```

```
sage: D = digraphs.Complete(9)
sage: H = D.subgraph([0, 1, 2]); H
Subgraph of (Complete digraph): Digraph on 3 vertices
sage: H = D.subgraph(edges=[(0, 1), (0, 2)])
sage: H.edges(labels=False)
[(0, 1), (0, 2)]
sage: H.vertices() == D.vertices()
True
sage: D
Complete digraph: Digraph on 9 vertices
sage: D.subgraph([0, 1, 2], inplace=True); D
Subgraph of (Complete digraph): Digraph on 3 vertices
sage: D.subgraph() == D
True
```

A more complicated example involving multiple edges and labels:

```
sage: G = Graph(multiedges=True, sparse=True)
sage: G.add_edges([(0, 1, 'a'), (0, 1, 'b'), (1, 0, 'c'), (0, 2, 'd'), (0, 2,
↪'e'), (2, 0, 'f'), (1, 2, 'g')])
sage: G.subgraph(edges=[(0, 1), (0, 2,'d'), (0, 2, 'not in graph')]).edges()
[(0, 1, 'a'), (0, 1, 'b'), (0, 1, 'c'), (0, 2, 'd')]
sage: J = G.subgraph(vertices=[0, 1], edges=[(0, 1, 'a'), (0, 2, 'c')])
sage: J.edges()
[(0, 1, 'a')]
sage: J.vertices()
[0, 1]
```

(continues on next page)

(continued from previous page)

```
sage: G.subgraph(vertices=G) == G
True
```

```
sage: D = DiGraph(multiedges=True, sparse=True)
sage: D.add_edges([(0, 1, 'a'), (0, 1, 'b'), (1, 0, 'c'), (0, 2, 'd'), (0, 2,
↪'e'), (2, 0, 'f'), (1, 2, 'g')])
sage: D.subgraph(edges=[(0, 1), (0, 2, 'd'), (0, 2, 'not in graph')]).edges()
[(0, 1, 'a'), (0, 1, 'b'), (0, 2, 'd')]
sage: H = D.subgraph(vertices=[0, 1], edges=[(0, 1, 'a'), (0, 2, 'c')])
sage: H.edges()
[(0, 1, 'a')]
sage: H.vertices()
[0, 1]
```

Using the property arguments:

```
sage: P = graphs.PetersenGraph()
sage: S = P.subgraph(vertex_property=lambda v: not (v % 2))
sage: S.vertices()
[0, 2, 4, 6, 8]
```

```
sage: C = graphs.CubeGraph(2)
sage: S = C.subgraph(edge_property=(lambda e: e[0][0] == e[1][0]))
sage: C.edges()
[('00', '01', None), ('00', '10', None), ('01', '11', None), ('10', '11',␣
↪None)]
sage: S.edges()
[('00', '01', None), ('10', '11', None)]
```

The algorithm is not specified, then a reasonable choice is made for speed:

```
sage: g = graphs.PathGraph(1000)
sage: g.subgraph(list(range(10))) # uses the 'add' algorithm
Subgraph of (Path graph): Graph on 10 vertices
```

subgraph_search(*G*, *induced=False*)

Return a copy of `G` in `self`.

INPUT:

- `G` – the (di)graph whose copy we are looking for in `self`
- `induced` – boolean (default: `False`); whether or not to search for an induced copy of `G` in `self`

OUTPUT:

If `induced=False`, return a copy of `G` in this graph. Otherwise, return an induced copy of `G` in `self`. If `G` is the empty graph, return the empty graph since it is a subgraph of every graph. Now suppose `G` is not the empty graph. If there is no copy (induced or otherwise) of `G` in `self`, we return `None`.

Note: The vertex labels and the edge labels in the graph are ignored.

See also:

- *subgraph_search_count()* – counts the number of copies of H inside of G
- *subgraph_search_iterator()* – iterator over the copies of H inside of G

ALGORITHM:

See the documentation of *SubgraphSearch*.

EXAMPLES:

The Petersen graph contains the path graph P_5:

```
sage: g = graphs.PetersenGraph()
sage: h1 = g.subgraph_search(graphs.PathGraph(5)); h1
Subgraph of (Petersen graph): Graph on 5 vertices
sage: h1.vertices(); h1.edges(labels=False)
[0, 1, 2, 3, 4]
[(0, 1), (1, 2), (2, 3), (3, 4)]
sage: I1 = g.subgraph_search(graphs.PathGraph(5), induced=True); I1
Subgraph of (Petersen graph): Graph on 5 vertices
sage: I1.vertices(); I1.edges(labels=False)
[0, 1, 2, 3, 8]
[(0, 1), (1, 2), (2, 3), (3, 8)]
```

It also contains the claw $K_{1,3}$:

```
sage: h2 = g.subgraph_search(graphs.ClawGraph()); h2
Subgraph of (Petersen graph): Graph on 4 vertices
sage: h2.vertices(); h2.edges(labels=False)
[0, 1, 4, 5]
[(0, 1), (0, 4), (0, 5)]
sage: I2 = g.subgraph_search(graphs.ClawGraph(), induced=True); I2
Subgraph of (Petersen graph): Graph on 4 vertices
sage: I2.vertices(); I2.edges(labels=False)
[0, 1, 4, 5]
[(0, 1), (0, 4), (0, 5)]
```

Of course the induced copies are isomorphic to the graphs we were looking for:

```
sage: I1.is_isomorphic(graphs.PathGraph(5))
True
sage: I2.is_isomorphic(graphs.ClawGraph())
True
```

However, the Petersen graph does not contain a subgraph isomorphic to K_3:

```
sage: g.subgraph_search(graphs.CompleteGraph(3)) is None
True
```

Nor does it contain a nonempty induced subgraph isomorphic to P_6:

```
sage: g.subgraph_search(graphs.PathGraph(6), induced=True) is None
True
```

The empty graph is a subgraph of every graph:

```
sage: g.subgraph_search(graphs.EmptyGraph())
Graph on 0 vertices
sage: g.subgraph_search(graphs.EmptyGraph(), induced=True)
Graph on 0 vertices
```

The subgraph may just have edges missing:

```
sage: k3 = graphs.CompleteGraph(3); p3 = graphs.PathGraph(3)
sage: k3.relabel(list('abc'))
sage: s = k3.subgraph_search(p3)
sage: s.edges(labels=False)
[('a', 'b'), ('b', 'c')]
```

Of course, P_3 is not an induced subgraph of K_3, though:

```
sage: k3 = graphs.CompleteGraph(3); p3 = graphs.PathGraph(3)
sage: k3.relabel(list('abc'))
sage: k3.subgraph_search(p3, induced=True) is None
True
```

If the graph has labels, the labels are just ignored:

```
sage: g.set_vertex(0, 'foo')
sage: c = g.subgraph_search(graphs.PathGraph(5))
sage: c.get_vertices()
{0: 'foo', 1: None, 2: None, 3: None, 4: None}
```

subgraph_search_count(*G*, *induced=False*)

Return the number of labelled occurrences of `G` in `self`.

INPUT:

- `G` – the (di)graph whose copies we are looking for in `self`
- `induced` – boolean (default: `False`); whether or not to count induced copies of `G` in `self`

Note: The vertex labels and the edge labels in the graph are ignored.

ALGORITHM:

See the documentation of *SubgraphSearch*.

See also:

- *subgraph_search()* – finds an subgraph isomorphic to H inside of a graph G
- *subgraph_search_iterator()* – iterator over the copies of a graph H inside of a graph G

EXAMPLES:

Counting the number of paths P_5 in a PetersenGraph:

```
sage: g = graphs.PetersenGraph()
sage: g.subgraph_search_count(graphs.PathGraph(5))
240
```

Requiring these subgraphs be induced:

```
sage: g.subgraph_search_count(graphs.PathGraph(5), induced=True)
120
```

If we define the graph T_k (the transitive tournament on k vertices) as the graph on $\{0, ..., k-1\}$ such that $ij \in T_k$ iif $i < j$, how many directed triangles can be found in T_5 ? The answer is of course 0:

```
sage: T5 = digraphs.TransitiveTournament(5)
sage: T5.subgraph_search_count(digraphs.Circuit(3))
0
```

If we count instead the number of T_3 in T_5, we expect the answer to be $\binom{5}{3}$:

```
sage: T3 = digraphs.TransitiveTournament(3)
sage: T5.subgraph_search_count(T3)
10
sage: binomial(5,3)
10
sage: T3.is_isomorphic(T5.subgraph(vertices=[0, 1, 2]))
True
```

The empty graph is a subgraph of every graph:

```
sage: g.subgraph_search_count(graphs.EmptyGraph())
1
```

If the graph has vertex labels or edge labels, the label is just ignored:

```
sage: g.set_vertex(0, 'foo')
sage: g.subgraph_search_count(graphs.PathGraph(5))
240
```

subgraph_search_iterator(*G*, *induced=False*)

Return an iterator over the labelled copies of `G` in `self`.

INPUT:

- `G` – the graph whose copies we are looking for in `self`
- `induced` – boolean (default: `False`); whether or not to iterate over the induced copies of `G` in `self`

Note: The vertex labels and the edge labels in the graph are ignored.

ALGORITHM:

See the documentation of *SubgraphSearch*.

OUTPUT:

Iterator over the labelled copies of `G` in `self`, as *lists*. For each value $(v_1, v_2, ..., v_k)$ returned, the first vertex of G is associated with v_1, the second with v_2, etc.

Note: This method also works on digraphs.

See also:

- *subgraph_search()* – finds an subgraph isomorphic to H inside of G
- *subgraph_search_count()* – counts the number of copies of H inside of G

EXAMPLES:

Iterating through all the labelled P_3 of P_5:

```
sage: g = graphs.PathGraph(5)
sage: for p in g.subgraph_search_iterator(graphs.PathGraph(3)):
....:     print(p)
[0, 1, 2]
[1, 2, 3]
[2, 1, 0]
[2, 3, 4]
[3, 2, 1]
[4, 3, 2]
```

If the graph has vertex labels or edge labels, the label is just ignored:

```
sage: g.set_vertex(0, 'foo')
sage: for p in g.subgraph_search_iterator(graphs.PathGraph(3)):
....:     print(p)
[0, 1, 2]
[1, 2, 3]
[2, 1, 0]
[2, 3, 4]
[3, 2, 1]
[4, 3, 2]
```

szeged_index()

Return the Szeged index of the graph.

For any $uv \in E(G)$, let $N_u(uv) = \{w \in G : d(u,w) < d(v,w)\}, n_u(uv) = |N_u(uv)|$

The Szeged index of a connected graph is then defined as [1]: $\sum_{uv \in E(G)} n_u(uv) \times n_v(uv)$

See the Wikipedia article Szeged_index for more details.

EXAMPLES:

True for any connected graph [1]:

```
sage: g=graphs.PetersenGraph()
sage: g.wiener_index()<= g.szeged_index()
True
```

True for all trees [1]:

```
sage: g=Graph()
sage: g.add_edges(graphs.CubeGraph(5).min_spanning_tree())
sage: g.wiener_index() == g.szeged_index()
True
```

REFERENCE:

[1] Klavzar S., Rajapakse A., Gutman I. (1996). The Szeged and the Wiener index of graphs. Applied Mathematics Letters, 9 (5), pp. 45-49.

tensor_product(*other*)

Return the tensor product of `self` and `other`.

The tensor product of G and H is the graph L with vertex set $V(L)$ equal to the Cartesian product of the vertices $V(G)$ and $V(H)$, and $((u,v),(w,x))$ is an edge iff - (u,w) is an edge of self, and - (v,x) is an edge of other.

The tensor product is also known as the categorical product and the kronecker product (refering to the kronecker matrix product). See the Wikipedia article Kronecker_product.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: T = C.tensor_product(Z); T
Graph on 10 vertices
sage: T.size()
10
sage: T.plot() # long time
Graphics object consisting of 21 graphics primitives
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: T = D.tensor_product(P); T
Graph on 200 vertices
sage: T.size()
900
sage: T.plot() # long time
Graphics object consisting of 1101 graphics primitives
```

to_dictionary(*edge_labels=False*, *multiple_edges=False*)

Return the graph as a dictionary.

INPUT:

- `edge_labels` – boolean (default: `False`); whether to include edge labels in the output
- `multiple_edges` – boolean (default: `False`); whether to include multiple edges in the output

OUTPUT:

The output depends on the input:

- If `edge_labels == False` and `multiple_edges == False`, the output is a dictionary associating to each vertex the list of its neighbors.
- If `edge_labels == False` and `multiple_edges == True`, the output is a dictionary the same as previously with one difference: the neighbors are listed with multiplicity.
- If `edge_labels == True` and `multiple_edges == False`, the output is a dictionary associating to each vertex u [a dictionary associating to each vertex v incident to u the label of edge (u, v)].
- If `edge_labels == True` and `multiple_edges == True`, the output is a dictionary associating to each vertex u [a dictionary associating to each vertex v incident to u [the list of labels of all edges between u and v]].

Note: When used on directed graphs, the explanations above can be understood by replacing the word "neighbors" by "out-neighbors"

EXAMPLES:

```
sage: g = graphs.PetersenGraph().to_dictionary()
sage: [(key, sorted(g[key])) for key in g]
[(0, [1, 4, 5]),
 (1, [0, 2, 6]),
 (2, [1, 3, 7]),
 (3, [2, 4, 8]),
 (4, [0, 3, 9]),
```

(continues on next page)

(continued from previous page)

```
 (5, [0, 7, 8]),
 (6, [1, 8, 9]),
 (7, [2, 5, 9]),
 (8, [3, 5, 6]),
 (9, [4, 6, 7])]
sage: graphs.PetersenGraph().to_dictionary(multiple_edges=True)
{0: [1, 4, 5], 1: [0, 2, 6],
 2: [1, 3, 7], 3: [2, 4, 8],
 4: [0, 3, 9], 5: [0, 7, 8],
 6: [1, 8, 9], 7: [2, 5, 9],
 8: [3, 5, 6], 9: [4, 6, 7]}
sage: graphs.PetersenGraph().to_dictionary(edge_labels=True)
{0: {1: None, 4: None, 5: None},
 1: {0: None, 2: None, 6: None},
 2: {1: None, 3: None, 7: None},
 3: {2: None, 4: None, 8: None},
 4: {0: None, 3: None, 9: None},
 5: {0: None, 7: None, 8: None},
 6: {1: None, 8: None, 9: None},
 7: {2: None, 5: None, 9: None},
 8: {3: None, 5: None, 6: None},
 9: {4: None, 6: None, 7: None}}
sage: graphs.PetersenGraph().to_dictionary(edge_labels=True,multiple_
↪edges=True)
{0: {1: [None], 4: [None], 5: [None]},
 1: {0: [None], 2: [None], 6: [None]},
 2: {1: [None], 3: [None], 7: [None]},
 3: {2: [None], 4: [None], 8: [None]},
 4: {0: [None], 3: [None], 9: [None]},
 5: {0: [None], 7: [None], 8: [None]},
 6: {1: [None], 8: [None], 9: [None]},
 7: {2: [None], 5: [None], 9: [None]},
 8: {3: [None], 5: [None], 6: [None]},
 9: {4: [None], 6: [None], 7: [None]}}
```

to_simple(*to_undirected=True*, *keep_label='any'*, *immutable=None*)

Return a simple version of the `self`.

In particular, loops and multiple edges are removed, and the graph might optionally be converted to an undirected graph.

INPUT:

- `to_undirected` – boolean (default: `True`); if `True`, the graph is also converted to an undirected graph
- `keep_label` – string (default: `'any'`); if there are multiple edges with different labels, this variable defines which label should be kept:
  - `'any'` – any label
  - `'min'` – the smallest label
  - `'max'` – the largest label

Warning: `'min'` and `'max'` only works if the labels can be compared. A `TypeError` might be raised when working with non-comparable objects in Python 3.

- `immutable` – boolean (default: `None`); whether to create a mutable/immutable copy. `immutable=None` (default) means that the graph and its copy will behave the same way.

EXAMPLES:

```
sage: G = DiGraph(loops=True, multiedges=True, sparse=True)
sage: G.add_edges([(0, 0, None), (1, 1, None), (2, 2, None), (2, 3, 1), (2, 3,
↪ 2), (3, 2, None)])
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (2, 3), (3, 2)]
sage: H = G.to_simple()
sage: H.edges(labels=False)
[(2, 3)]
sage: H.is_directed()
False
sage: H.allows_loops()
False
sage: H.allows_multiple_edges()
False
sage: G.to_simple(to_undirected=False, keep_label='min').edges()
[(2, 3, 1), (3, 2, None)]
sage: G.to_simple(to_undirected=False, keep_label='max').edges()
[(2, 3, 2), (3, 2, None)]
```

transitive_closure(*loops=True*)

Return the transitive closure of the (di)graph.

The transitive closure of a graph G has an edge (x, y) if and only if there is a path between x and y in G.

The transitive closure of any (strongly) connected component of a (di)graph is a complete graph. The transitive closure of a directed acyclic graph is a directed acyclic graph representing the full partial order.

Note: If the (di)graph allows loops, its transitive closure will by default have one loop edge per vertex. This can be prevented by disallowing loops in the (di)graph (`self.allow_loops(False)`).

EXAMPLES:

```
sage: g = graphs.PathGraph(4)
sage: g.transitive_closure()
Transitive closure of Path graph: Graph on 4 vertices
sage: g.transitive_closure().is_isomorphic(graphs.CompleteGraph(4))
True
sage: g = DiGraph({0: [1, 2], 1: [3], 2: [4, 5]})
sage: g.transitive_closure().edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 3), (2, 4), (2, 5)]
```

On an immutable digraph:

```
sage: digraphs.Path(5).copy(immutable=True).transitive_closure()
Transitive closure of Path: Digraph on 5 vertices
```

The transitive closure of a (di)graph allowing loops has by default a loop edge per vertex. Parameter `loops` allows to prevent that:

```
sage: G = digraphs.Circuit(3)
sage: G.transitive_closure().loop_edges(labels=False)
```

(continues on next page)

(continued from previous page)

```
[]
sage: G.allow_loops(True)
sage: G.transitive_closure().loop_edges(labels=False)
[(0, 0), (1, 1), (2, 2)]
```

```
sage: G = graphs.CycleGraph(3)
sage: G.transitive_closure().loop_edges(labels=False)
[]
sage: G.allow_loops(True)
sage: G.transitive_closure().loop_edges(labels=False)
[(0, 0), (1, 1), (2, 2)]
```

transitive_reduction()

Return a transitive reduction of a graph.

A transitive reduction H of G has a path from x to y if and only if there was a path from x to y in G. Deleting any edge of H destroys this property. A transitive reduction is not unique in general. A transitive reduction has the same transitive closure as the original graph.

A transitive reduction of a complete graph is a tree. A transitive reduction of a tree is itself.

EXAMPLES:

```
sage: g = graphs.PathGraph(4)
sage: g.transitive_reduction() == g
True
sage: g = graphs.CompleteGraph(5)
sage: h = g.transitive_reduction(); h.size()
4
sage: g = DiGraph({0: [1, 2], 1: [2, 3, 4, 5], 2: [4, 5]})
sage: g.transitive_reduction().size()
5
```

traveling_salesman_problem(*use_edge_labels=False*, *maximize=False*, *solver=None*, *constraint_generation=None*, *verbose=0*, *verbose_constraints=False*)

Solve the traveling salesman problem (TSP)

Given a graph (resp. a digraph) G with weighted edges, the traveling salesman problem consists in finding a Hamiltonian cycle (resp. circuit) of the graph of minimum cost.

This TSP is one of the most famous NP-Complete problems, this function can thus be expected to take some time before returning its result.

INPUT:

- `use_edge_labels` – boolean (default: `False`); whether to solve the weighted traveling salesman problem where the weight of an edge is defined by its label (a label set to `None` or `{}` being considered as a weight of 1), or the non-weighted version (i.e., the Hamiltonian cycle problem)
- `maximize` – boolean (default: `False`); whether to compute a minimum (default) or a maximum (when `maximize == True`) weight tour (or Hamiltonian cycle). This parameter is considered only if `use_edge_labels == True`.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.

- `constraint_generation` – boolean (default: `None`); whether to use constraint generation when solving the Mixed Integer Linear Program.

  When `constraint_generation = None`, constraint generation is used whenever the graph has a density larger than 70%.

- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.
- `verbose_constraints` – boolean (default: `False`); whether to display which constraints are being generated

OUTPUT:

A solution to the TSP, as a `Graph` object whose vertex set is $V(G)$, and whose edges are only those of the solution.

ALGORITHM:

This optimization problem is solved through the use of Linear Programming.

Note: This function is correctly defined for both graph and digraphs. In the second case, the returned cycle is a circuit of optimal cost.

EXAMPLES:

The Heawood graph is known to be Hamiltonian:

```
sage: g = graphs.HeawoodGraph()
sage: tsp = g.traveling_salesman_problem()
sage: tsp
TSP from Heawood graph: Graph on 14 vertices
```

The solution to the TSP has to be connected:

```
sage: tsp.is_connected()
True
```

It must also be a 2-regular graph:

```
sage: tsp.is_regular(k=2)
True
```

And obviously it is a subgraph of the Heawood graph:

```
sage: tsp.is_subgraph(g, induced=False)
True
```

On the other hand, the Petersen Graph is known not to be Hamiltonian:

```
sage: g = graphs.PetersenGraph()
sage: tsp = g.traveling_salesman_problem()
Traceback (most recent call last):
...
EmptySetError: the given graph is not Hamiltonian
```

One easy way to change it is obviously to add to this graph the edges corresponding to a Hamiltonian cycle. If we do this by setting the cost of these new edges to 2, while the others are set to 1, we notice that not all the edges we added are used in the optimal solution

```
sage: for u, v in g.edge_iterator(labels=None):
....:     g.set_edge_label(u, v, 1)

sage: cycle = graphs.CycleGraph(10)
sage: for u,v in cycle.edges(labels=None, sort=False):
....:     if not g.has_edge(u, v):
....:         g.add_edge(u, v)
....:     g.set_edge_label(u, v, 2)

sage: tsp = g.traveling_salesman_problem(use_edge_labels=True)
sage: sum( tsp.edge_labels() ) < 2 * 10
True
```

If we pick $1/2$ instead of 2 as a cost for these new edges, they clearly become the optimal solution:

```
sage: for u, v in cycle.edges(labels=None, sort=False):
....:     g.set_edge_label(u,v,1/2)

sage: tsp = g.traveling_salesman_problem(use_edge_labels=True)
sage: sum(tsp.edge_labels()) == (1/2) * 10
True
```

Search for a minimum and a maximum weight Hamiltonian cycle:

```
sage: G = Graph([(0, 1, 1), (0, 2, 2), (0, 3, 1), (1, 2, 1), (1, 3, 2), (2, 3,
↪ 1)])
sage: tsp = G.traveling_salesman_problem(use_edge_labels=True, maximize=False)
sage: print(sum(tsp.edge_labels()))
4
sage: tsp = G.traveling_salesman_problem(use_edge_labels=True, maximize=True)
sage: print(sum(tsp.edge_labels()))
6
```

triangles_count(*algorithm=None*)

Return the number of triangles in the (di)graph.

For digraphs, we count the number of directed circuit of length 3.

INPUT:

- `algorithm` – string (default: `None`); specifies the algorithm to use (note that only `'iter'` is available for directed graphs):
  - `'sparse_copy'` – counts the triangles in a sparse copy of the graph (see *sage.graphs.base.static_sparse_graph*). Calls *static_sparse_graph.triangles_count*
  - `'dense_copy'` – counts the triangles in a dense copy of the graph (see *sage.graphs.base.static_dense_graph*). Calls *static_dense_graph.triangles_count*
  - `'matrix'` uses the trace of the cube of the adjacency matrix
  - `'iter'` iterates over the pairs of neighbors of each vertex. No copy of the graph is performed
  - `None` – for undirected graphs, uses `"sparse_copy"` or `"dense_copy"` depending on whether the graph is stored as dense or sparse. For directed graphs, uses `'iter'`.

EXAMPLES:

The Petersen graph is triangle free and thus:

```
sage: G = graphs.PetersenGraph()
sage: G.triangles_count()
0
```

Any triple of vertices in the complete graph induces a triangle so we have:

```
sage: G = graphs.CompleteGraph(15)
sage: G.triangles_count() == binomial(15, 3)
True
```

The 2-dimensional DeBruijn graph of 2 symbols has 2 directed C_3:

```
sage: G = digraphs.DeBruijn(2,2)
sage: G.triangles_count()
2
```

The directed n-cycle is trivially triangle free for $n > 3$:

```
sage: G = digraphs.Circuit(10)
sage: G.triangles_count()
0
```

union(*other*, *immutable=None*)

Return the union of `self` and `other`.

If the graphs have common vertices, the common vertices will be identified.

If one of the two graphs allows loops (or multiple edges), the resulting graph will allow loops (or multiple edges).

If both graphs are weighted the resulting graphs is weighted.

If both graphs are immutable, the resulting graph is immutable, unless requested otherwise.

INPUT:

- `immutable` – boolean (default: `None`); whether to create a mutable/immutable union. `immutable=None` (default) means that the graphs and their union will behave the same way.

See also:

- *disjoint_union()*
- *join()*

EXAMPLES:

```
sage: G = graphs.CycleGraph(3)
sage: H = graphs.CycleGraph(4)
sage: J = G.union(H); J
Graph on 4 vertices
sage: J.vertices()
[0, 1, 2, 3]
sage: J.edges(labels=False)
[(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)]
```

vertex_boundary(*vertices1*, *vertices2=None*)

Return a list of all vertices in the external boundary of `vertices1`, intersected with `vertices2`.

If `vertices2` is `None`, then `vertices2` is the complement of `vertices1`. This is much faster if `vertices1` is smaller than `vertices2`.

The external boundary of a set of vertices is the union of the neighborhoods of each vertex in the set. Note that in this implementation, since `vertices2` defaults to the complement of `vertices1`, if a vertex v has a loop, then `vertex_boundary(v)` will not contain v.

In a digraph, the external boundary of a vertex v are those vertices u with an arc (v, u).

EXAMPLES:

```
sage: G = graphs.CubeGraph(4)
sage: l = ['0111', '0000', '0001', '0011', '0010', '0101', '0100', '1111',
↪'1101', '1011', '1001']
sage: sorted(G.vertex_boundary(['0000', '1111'], l))
['0001', '0010', '0100', '0111', '1011', '1101']
```

```
sage: D = DiGraph({0: [1, 2], 3: [0]})
sage: D.vertex_boundary([0])
[1, 2]
```

vertex_connectivity(*G*, *value_only=True*, *sets=False*, *k=None*, *solver=None*, *verbose=0*)

Return the vertex connectivity of the graph.

For more information, see the Wikipedia article Connectivity_(graph_theory) and the Wikipedia article K-vertex-connected_graph.

Note:

- When the graph is directed, this method actually computes the *strong* connectivity, (i.e. a directed graph is strongly k-connected if there are k vertex disjoint paths between any two vertices u, v). If you do not want to consider strong connectivity, the best is probably to convert your `DiGraph` object to a `Graph` object, and compute the connectivity of this other graph.
- By convention, a complete graph on n vertices is $n - 1$ connected. In this case, no certificate can be given as there is no pair of vertices split by a cut of order $k - 1$. For this reason, the certificates returned in this situation are empty.

INPUT:

- `G` – the input Sage (Di)Graph
- `value_only` – boolean (default: `True`)
  - When set to `True` (default), only the value is returned.
  - When set to `False`, both the value and a minimum vertex cut are returned.
- **`sets` – boolean (default: `False`); whether to also return the two** sets of vertices that are disconnected by the cut (implies `value_only=False`)
- `k` – integer (default: `None`); when specified, check if the vertex connectivity of the (di)graph is larger or equal to k. The method thus outputs a boolean only.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers, see the method `solve` of the class `MixedIntegerLinearProgram`. Use method `sage.numerical.backends.generic_backend.default_mip_solver()` to know which default solver is used or to set the default solver.

- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic application on a `PappusGraph`:

```
sage: from sage.graphs.connectivity import vertex_connectivity
sage: g=graphs.PappusGraph()
sage: vertex_connectivity(g)
3
sage: g.vertex_connectivity()
3
```

In a grid, the vertex connectivity is equal to the minimum degree, in which case one of the two sets is of cardinality 1:

```
sage: g = graphs.GridGraph([ 3,3 ])
sage: [value, cut, [ setA, setB ]] = vertex_connectivity(g, sets=True)
sage: len(setA) == 1 or len(setB) == 1
True
```

A vertex cut in a tree is any internal vertex:

```
sage: tree = graphs.RandomTree(15)
sage: val, [cut_vertex] = vertex_connectivity(tree, value_only=False)
sage: tree.degree(cut_vertex) > 1
True
```

When `value_only = True`, this function is optimized for small connectivity values and does not need to build a linear program.

It is the case for connected graphs which are not connected:

```
sage: g = 2 * graphs.PetersenGraph()
sage: vertex_connectivity(g)
0
```

Or if they are just 1-connected:

```
sage: g = graphs.PathGraph(10)
sage: vertex_connectivity(g)
1
```

For directed graphs, the strong connectivity is tested through the dedicated function:

```
sage: g = digraphs.ButterflyGraph(3)
sage: vertex_connectivity(g)
0
```

A complete graph on 10 vertices is 9-connected:

```
sage: g = graphs.CompleteGraph(10)
sage: vertex_connectivity(g)
9
```

A complete digraph on 10 vertices is 9-connected:

```
sage: g = DiGraph(graphs.CompleteGraph(10))
sage: vertex_connectivity(g)
9
```

When parameter `k` is set, we only check for the existence of a vertex cut of order at least `k`:

```
sage: g = graphs.PappusGraph()
sage: vertex_connectivity(g, k=3)
True
sage: vertex_connectivity(g, k=4)
False
```

vertex_cut(*s*, *t*, *value_only=True*, *vertices=False*, *solver=None*, *verbose=0*)

Return a minimum vertex cut between non-adjacent vertices s and t represented by a list of vertices.

A vertex cut between two non-adjacent vertices is a set U of vertices of `self` such that the graph obtained by removing U from `self` is disconnected. For more information, see the Wikipedia article Cut_(graph_theory).

INPUT:

- `value_only` – boolean (default: `True`); whether to return only the size of the minimum cut, or to also return the set U of vertices of the cut
- `vertices` – boolean (default: `False`); whether to also return the two sets of vertices that are disconnected by the cut. Implies `value_only` set to `False`.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

Real number or tuple, depending on the given arguments (examples are given below).

EXAMPLES:

A basic application in the Pappus graph:

```
sage: g = graphs.PappusGraph()
sage: g.vertex_cut(1, 16, value_only=True)
3
```

In the bipartite complete graph $K_{2,8}$, a cut between the two vertices in the size 2 part consists of the other 8 vertices:

```
sage: g = graphs.CompleteBipartiteGraph(2, 8)
sage: [value, vertices] = g.vertex_cut(0, 1, value_only=False)
sage: print(value)
8
sage: vertices == list(range(2, 10))
True
```

Clearly, in this case the two sides of the cut are singletons:

```
sage: [value, vertices, [set1, set2]] = g.vertex_cut(0, 1, vertices=True)
sage: len(set1) == 1
True
```

(continues on next page)

(continued from previous page)

```
sage: len(set2) == 1
True
```

vertex_disjoint_paths(*s*, *t*, *solver=None*, *verbose=0*)

Return a list of vertex-disjoint paths between two vertices.

The vertex version of Menger's theorem asserts that the size of the minimum vertex cut between two vertices s and t (the minimum number of vertices whose removal disconnects s and t) is equal to the maximum number of pairwise vertex-independent paths from s to t.

This function returns a list of such paths.

INPUT:

- `s,t` – two vertices of the graph.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

In a complete bipartite graph

```
sage: g = graphs.CompleteBipartiteGraph(2, 3)
sage: g.vertex_disjoint_paths(0, 1)
[[0, 2, 1], [0, 3, 1], [0, 4, 1]]
```

vertex_iterator(*vertices=None*)

Return an iterator over the given vertices.

Returns `False` if not given a vertex, sequence, iterator or `None`. `None` is equivalent to a list of every vertex. Note that `for v in G` syntax is allowed.

INPUT:

- `vertices` – iterated vertices are these intersected with the vertices of the (di)graph

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: for v in P.vertex_iterator():
....:     print(v)
0
1
2
...
8
9
```

```
sage: G = graphs.TetrahedralGraph()
sage: for i in G:
....:     print(i)
0
1
2
3
```

Note that since the intersection option is available, the vertex_iterator() function is sub-optimal, speed-wise, but note the following optimization:

```
sage: timeit V = P.vertices()                    # not tested
100000 loops, best of 3: 8.85 [micro]s per loop
sage: timeit V = list(P.vertex_iterator())       # not tested
100000 loops, best of 3: 5.74 [micro]s per loop
```

vertices(*sort=True*, *key=None*)

Return a list of the vertices.

INPUT:

- `sort` – boolean (default: `True`); if `True`, vertices are sorted according to the default ordering
- `key` – a function (default: `None`); a function that takes a vertex as its one argument and returns a value that can be used for comparisons in the sorting algorithm (we must have `sort=True`)

OUTPUT:

The list of vertices of the (di)graph.

Warning: Since any object may be a vertex, there is no guarantee that any two vertices will be comparable. With default objects for vertices (all integers), or when all the vertices are of the same simple type, then there should not be a problem with how the vertices will be sorted. However, if you need to guarantee a total order for the sorting of the edges, use the `key` argument, as illustrated in the examples below.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If you do not care about sorted output and you are concerned about the time taken to sort, consider the following alternative:

```
sage: timeit V = P.vertices()                    # not tested
625 loops, best of 3: 3.86 [micro]s per loop
sage: timeit V = P.vertices(sort=False)          # not tested
625 loops, best of 3: 2.06 [micro]s per loop
sage: timeit V = list(P.vertex_iterator())       # not tested
625 loops, best of 3: 2.05 [micro]s per loop
sage: timeit('V = list(P)')                      # not tested
625 loops, best of 3: 1.98 [micro]s per loop
```

We illustrate various ways to use a `key` to sort the list:

```
sage: H = graphs.HanoiTowerGraph(3, 3, labels=False)
sage: H.vertices()
[0, 1, 2, 3, 4, ... 22, 23, 24, 25, 26]
sage: H.vertices(key=lambda x: -x)
[26, 25, 24, 23, 22, ... 4, 3, 2, 1, 0]
```

```
sage: G = graphs.HanoiTowerGraph(3, 3)
sage: G.vertices()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), ... (2, 2, 1), (2, 2, 2)]
```

(continues on next page)

(continued from previous page)

```
sage: G.vertices(key = lambda x: (x[1], x[2], x[0]))
[(0, 0, 0), (1, 0, 0), (2, 0, 0), (0, 0, 1), ... (1, 2, 2), (2, 2, 2)]
```

The discriminant of a polynomial is a function that returns an integer. We build a graph whose vertices are polynomials, and use the discriminant function to provide an ordering. Note that since functions are first-class objects in Python, we can specify precisely the function from the Sage library that we wish to use as the key:

```
sage: t = polygen(QQ, 't')
sage: K = Graph({5*t: [t^2], t^2: [t^2+2], t^2+2: [4*t^2-6], 4*t^2-6: [5*t]})
sage: dsc = sage.rings.polynomial.polynomial_rational_flint.Polynomial_
↪rational_flint.discriminant
sage: verts = K.vertices(key=dsc)
sage: verts
[t^2 + 2, t^2, 5*t, 4*t^2 - 6]
sage: [x.discriminant() for x in verts]
[-8, 0, 1, 96]
```

weighted(*new=None*)

Whether the (di)graph is to be considered as a weighted (di)graph.

INPUT:

- `new` – boolean (default: `None`); if it is provided, then the weightedness flag is set accordingly. This is not allowed for immutable graphs.

Note: Changing the weightedness flag changes the ==-class of a graph and is thus not allowed for immutable graphs.

Edge weightings can still exist for (di)graphs `G` where `G.weighted()` is `False`.

EXAMPLES:

Here we have two graphs with different labels, but `weighted()` is `False` for both, so we just check for the presence of edges:

```
sage: G = Graph({0: {1: 'a'}}, sparse=True)
sage: H = Graph({0: {1: 'b'}}, sparse=True)
sage: G == H
True
```

Now one is weighted and the other is not, and thus the graphs are not equal:

```
sage: G.weighted(True)
sage: H.weighted()
False
sage: G == H
False
```

However, if both are weighted, then we finally compare ‘a’ to ‘b’:

```
sage: H.weighted(True)
sage: G == H
False
```

weighted_adjacency_matrix(*sparse=True*, *vertices=None*)

Return the weighted adjacency matrix of the graph.

By default, each vertex is represented by its position in the list returned by method *vertices()*.

INPUT:

- sparse – boolean (default: True); whether to use a sparse or a dense matrix
- vertices – list (default: None); when specified, each vertex is represented by its position in the list vertices, otherwise each vertex is represented by its position in the list returned by method *vertices()*

EXAMPLES:

```
sage: G = Graph(sparse=True, weighted=True)
sage: G.add_edges([(0, 1, 1), (1, 2, 2), (0, 2, 3), (0, 3, 4)])
sage: M = G.weighted_adjacency_matrix(); M
[0 1 3 4]
[1 0 2 0]
[3 2 0 0]
[4 0 0 0]
sage: H = Graph(data=M, format='weighted_adjacency_matrix', sparse=True)
sage: H == G
True
sage: G.weighted_adjacency_matrix(vertices=[3, 2, 1, 0])
[0 0 0 4]
[0 0 2 3]
[0 2 0 1]
[4 3 1 0]
```

wiener_index(*by_weight=False*, *algorithm=None*, *weight_function=None*, *check_weight=True*)

Return the Wiener index of the graph.

The graph is expected to have no cycles of negative weight.

The Wiener index of a graph G is $W(G) = \frac{1}{2}\sum_{u,v\in G} d(u,v)$ where $d(u,v)$ denotes the distance between vertices u and v (see [KRG1996]).

For more information on the input variables and more examples, we refer to *shortest_paths()* and *shortest_path_all_pairs()*, which have very similar input variables.

INPUT:

- by_weight – boolean (default: False); if True, the edges in the graph are weighted, otherwise all edges have weight 1
- algorithm – string (default: None); one of the following algorithms:
  - For by_weight==False only:
    - 'BFS' - the computation is done through a BFS centered on each vertex successively.
    - 'Floyd-Warshall-Cython' - the Cython implementation of the Floyd-Warshall algorithm. Usually slower than 'BFS'.
  - For graphs without negative weights:
    - 'Dijkstra_Boost': the Dijkstra algorithm, implemented in Boost.
    - 'Dijkstra_NetworkX': the Dijkstra algorithm, implemented in NetworkX. Usually slower than 'Dijkstra_Boost'.
  - For graphs with negative weights:
    - 'Johnson_Boost': the Johnson algorithm, implemented in Boost.

* 'Floyd-Warshall-Python' - the Python implementation of the Floyd-Warshall algorithm. Usually slower than 'Johnson_Boost'.

– None (default): Sage chooses the best algorithm: 'BFS' for unweighted graphs, 'Dijkstra_Boost' if all weights are positive, 'Johnson_Boost', otherwise.

- weight_function – function (default: None); a function that takes as input an edge (u, v, l) and outputs its weight. If not None, by_weight is automatically set to True. If None and by_weight is True, we use the edge label l as a weight.
- check_weight – boolean (default: True); if True, we check that the weight_function outputs a number for each edge

EXAMPLES:

```
sage: G = Graph( { 0: {1: None}, 1: {2: None}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2}
↪ }, sparse=True)
sage: G.wiener_index()
15
sage: G.wiener_index(weight_function=lambda e:(e[2] if e[2] is not None else␣
↪1))
20
sage: G.wiener_index(weight_function=lambda e:(e[2] if e[2] is not None else␣
↪200))
820
sage: G.wiener_index(algorithm='BFS')
15
sage: G.wiener_index(algorithm='Floyd-Warshall-Cython')
15
sage: G.wiener_index(algorithm='Floyd-Warshall-Python')
15
sage: G.wiener_index(algorithm='Dijkstra_Boost')
15
sage: G.wiener_index(algorithm='Johnson_Boost')
15
sage: G.wiener_index(algorithm='Dijkstra_NetworkX')
15
```

sage.graphs.generic_graph.**graph_isom_equivalent_non_edge_labeled_graph**(*g*, *partition=None*, *standard_label=None*, *return_relabeling=False*, *return_edge_labels=False*, *inplace=False*, *ignore_edge_labels=False*)

Helper function for canonical labeling of edge labeled (di)graphs.

Translates to a bipartite incidence-structure type graph appropriate for computing canonical labels of edge labeled and/or multi-edge graphs. Note that this is actually computationally equivalent to implementing a change on an inner loop of the main algorithm – namely making the refinement procedure sort for each label.

If the graph is a multigraph, it is translated to a non-multigraph, where each instance of multiple

edges is converted to a single edge labeled with a list `[[label1, multiplicity], [label2, multiplicity], ...]` describing how many edges of each label were originally there. Then in either case we are working on a graph without multiple edges. At this point, we create another (partially bipartite) graph, whose left vertices are the original vertices of the graph, and whose right vertices represent the labeled edges. Any unlabeled edges in the original graph are also present in the new graph, and – this is the bipartite aspect – for every labeled edge e from v to w in the original graph, there is an edge between the right vertex corresponding to e and each of the left vertices corresponding to v and w. We partition the left vertices as they were originally, and the right vertices by common labels: only automorphisms taking edges to like-labeled edges are allowed, and this additional partition information enforces this on the new graph.

INPUT:

- `g` – Graph or DiGraph
- `partition` – list (default: `None`); a partition of the vertices as a list of lists of vertices. If given, the partition of the vertices is as well relabeled
- `standard_label` – (default: `None`); edges in `g` with this label are preserved in the new graph
- `return_relabeling` – boolean (default: `False`); whether to return a dictionary containing the relabeling
- `return_edge_labels` – boolean (default: `False`); whether the different `edge_labels` are returned (useful if inplace is `True`)
- `inplace` – boolean (default: `False`); whether the input (di)graph `g` is modified or the return a new (di)graph. Note that attributes of `g` are *not* copied for speed issues, only edges and vertices.
- `ignore_edge_labels` – boolean (default: `False`): if `True`, ignore edge labels, so when constructing the new graph, only multiple edges are replaced with vertices. Labels on multiple edges are ignored – only the multiplicity is relevant, so multiple edges with the same multiplicity in the original graph correspond to right vertices in the same partition in the new graph.

OUTPUT:

- if `inplace` is `False`: the unlabeled graph without multiple edges
- the partition of the vertices
- if `return_relabeling` is `True`: a dictionary containing the relabeling
- if `return_edge_labels` is `True`: the list of (former) edge labels is returned

EXAMPLES:

```
sage: from sage.graphs.generic_graph import graph_isom_equivalent_non_edge_
↪labeled_graph

sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edges((0, 1, i) for i in range(10))
sage: G.add_edge(1,2,'string')
sage: G.add_edge(2,123)
sage: graph_isom_equivalent_non_edge_labeled_graph(G, partition=[[0,123],[1,2]])
[Graph on 6 vertices, [[1, 0], [2, 3], [5], [4]]]

sage: g, part = graph_isom_equivalent_non_edge_labeled_graph(G)
sage: g, sorted(part)
(Graph on 6 vertices, [[0, 1, 2, 3], [4], [5]])
sage: g.edges(sort=True)
[(0, 3, None), (1, 4, None), (2, 4, None), (2, 5, None), (3, 5, None)]

sage: g = graph_isom_equivalent_non_edge_labeled_graph(G,standard_label='string',
↪return_edge_labels=True)
```

(continues on next page)

(continued from previous page)

```
sage: g[0]
Graph on 6 vertices
sage: g[0].edges(sort=True)
[(0, 5, None), (1, 4, None), (2, 3, None), (2, 4, None), (3, 5, None)]
sage: g[1]
[[0, 1, 2, 3], [4], [5]]
sage: g[2]
[[['string', 1]], [[0, 1], [1, 1], [2, 1], [3, 1], [4, 1], [5, 1], [6, 1], [7, 1],
↪ [8, 1], [9, 1]], [[None, 1]]]

sage: graph_isom_equivalent_non_edge_labeled_graph(G, inplace=True)
[[[0, 1, 2, 3], [5], [4]]]
sage: G.edges(sort=True)
[(0, 3, None), (1, 4, None), (2, 4, None), (2, 5, None), (3, 5, None)]

sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edges((0, 1) for i in range(10))
sage: G.add_edge(1, 2, 'a')
sage: G.add_edge(1, 3, 'b')
sage: G.add_edge(2, 3, 'b')
sage: graph_isom_equivalent_non_edge_labeled_graph(G)[0]
Graph on 8 vertices
sage: graph_isom_equivalent_non_edge_labeled_graph(G, ignore_edge_labels=True)[0]
Graph on 5 vertices

sage: G = Graph(multiedges=True,sparse=True)
sage: G.add_edges((0, 1, i) for i in range(5))
sage: G.add_edges((0, 2, i+10) for i in range(5))
sage: G.add_edges((0, 3) for i in range(4))
sage: g0 = graph_isom_equivalent_non_edge_labeled_graph(G)
sage: g1 = graph_isom_equivalent_non_edge_labeled_graph(G, ignore_edge_
↪labels=True)
sage: g0
[Graph on 7 vertices, [[0, 1, 2, 3], [4], [5], [6]]]
sage: g1
[Graph on 7 vertices, [[0, 1, 2, 3], [6], [4, 5]]]
```

sage.graphs.generic_graph.**igraph_feature**()

sage.graphs.generic_graph.**tachyon_vertex_plot**(*g*, *bgcolor=(1, 1, 1)*, *vertex_colors=None*, *vertex_size=0.06*, *pos3d=None*, ***kwds*)

Helper function for plotting graphs in 3d with `Tachyon`.

Returns a plot containing only the vertices, as well as the 3d position dictionary used for the plot.

INPUT:

- `pos3d` – a 3D layout of the vertices
- various rendering options

EXAMPLES:

```
sage: G = graphs.TetrahedralGraph()
sage: from sage.graphs.generic_graph import tachyon_vertex_plot
sage: T,p = tachyon_vertex_plot(G, pos3d=G.layout(dim=3))
sage: type(T)
<class 'sage.plot.plot3d.tachyon.Tachyon'>
```

(continues on next page)

(continued from previous page)

```
sage: type(p)
<... 'dict'>
```

1.2 Undirected graphs

This module implements functions and operations involving undirected graphs.

Algorithmically hard stuff

chromatic_index()	Return the chromatic index of the graph.
chromatic_number()	Return the minimal number of colors needed to color the vertices of the graph.
chromatic_polynomial()	Compute the chromatic polynomial of the graph G.
chromatic_quasisymmetric_function()	Return the chromatic quasisymmetric function of self.
chromatic_symmetric_function()	Return the chromatic symmetric function of self.
coloring()	Return the first (optimal) proper vertex-coloring found.
convexity_properties()	Return a ConvexityProperties object corresponding to self.
has_homomorphism_to()	Checks whether there is a homomorphism between two graphs.
independent_set()	Return a maximum independent set.
independent_set_of_representatives()	Return an independent set of representatives.
is_perfect()	Tests whether the graph is perfect.
matching_polynomial()	Computes the matching polynomial of the graph G.
minor()	Return the vertices of a minor isomorphic to H in the current graph.
pathwidth()	Compute the pathwidth of self (and provides a decomposition)
rank_decomposition()	Compute an optimal rank-decomposition of the given graph.
topological_minor()	Return a topological H-minor from self if one exists.
treewidth()	Computes the tree-width of G (and provides a decomposition)
tutte_polynomial()	Return the Tutte polynomial of the graph G.
vertex_cover()	Return a minimum vertex cover of self represented by a set of vertices.

Basic methods

bipartite_color()	Return a dictionary with vertices as the keys and the color class as the values.
bipartite_sets()	Return (X, Y) where X and Y are the nodes in each bipartite set of graph G.
graph6_string()	Return the graph6 representation of the graph as an ASCII string.
is_directed()	Since graph is undirected, returns False.
join()	Return the join of self and other.
sparse6_string()	Return the sparse6 representation of the graph as an ASCII string.
to_directed()	Return a directed version of the graph.
to_undirected()	Since the graph is already undirected, simply returns a copy of itself.
write_to_eps()	Write a plot of the graph to filename in eps format.

Clique-related methods

all_cliques()	Iterator over the cliques in `graph`.
atoms_and_clique_separators()	Return the atoms of the decomposition of G by clique minimal separators.
clique_complex()	Return the clique complex of self.
clique_maximum()	Return the vertex set of a maximal order complete subgraph.
clique_number()	Return the order of the largest clique of the graph
clique_polynomial()	Return the clique polynomial of self.
cliques_containing_vertex()	Return the cliques containing each vertex, represented as a dictionary of lists of lists, keyed by vertex.
cliques_get_clique_bipartite()	Return a bipartite graph constructed such that maximal cliques are the right vertices and the left vertices are retained from the given graph. Right and left vertices are connected if the bottom vertex belongs to the clique represented by a top vertex.
cliques_get_max_clique_graph()	Return the clique graph.
cliques_maximal()	Return the list of all maximal cliques.
cliques_maximum()	Returns the vertex sets of *ALL* the maximum complete subgraphs.
cliques_number_of()	Return a dictionary of the number of maximal cliques containing each vertex, keyed by vertex.
cliques_vertex_clique_number()	Return a dictionary of sizes of the largest maximal cliques containing each vertex, keyed by vertex.

Connectivity, orientations, trees

bounded_outdegree_orientation()	Computes an orientation of `self` such that every vertex v has out-degree less than $b(v)$
bridges()	Return a list of the bridges (or cut edges).
cleave()	Return the connected subgraphs separated by the input vertex cut.
degree_constrained_subgraph()	Returns a degree-constrained subgraph.
ear_decomposition()	Return an Ear decomposition of the graph.
gomory_hu_tree()	Return a Gomory-Hu tree of self.
is_triconnected()	Check whether the graph is triconnected.
minimum_outdegree_orientation()	Returns an orientation of `self` with the smallest possible maximum outdegree.
orientations()	Return an iterator over orientations of `self`.
random_orientation()	Return a random orientation of a graph G.
random_spanning_tree()	Return a random spanning tree of the graph.
spanning_trees()	Returns a list of all spanning trees.
spqr_tree()	Return an SPQR-tree representing the triconnected components of the graph.
strong_orientation()	Returns a strongly connected orientation of the current graph.
strong_orientations_iterator()	Returns an iterator over all strong orientations of a graph G.

Distances

centrality_degree()	Return the degree centrality of a vertex.

Domination

is_dominating()	Check whether `dom` is a dominating set of `G`.
is_redundant()	Check whether `dom` has redundant vertices.
minimal_dominating_sets()	Return an iterator over the minimal dominating sets of a graph.
private_neighbors()	Return the private neighbors of a vertex with repect to other vertices.

Graph properties

apex_vertices()	Return the list of apex vertices.
is_apex()	Test if the graph is apex.
is_arc_transitive()	Check if self is an arc-transitive graph
is_asteroidal_triple_free()	Test if the input graph is asteroidal triple-free
is_biconnected()	Test if the graph is biconnected.
is_block_graph()	Return whether this graph is a block graph.
is_cactus()	Check whether the graph is cactus graph.
is_cartesian_product()	Test whether the graph is a Cartesian product.
is_circumscribable()	Test whether the graph is the graph of a circumscribed polyhedron.
is_cograph()	Check whether the graph is cograph.
is_comparability()	Tests whether the graph is a comparability graph
is_distance_regular()	Test if the graph is distance-regular
is_edge_transitive()	Check if self is an edge transitive graph.
is_even_hole_free()	Tests whether self contains an induced even hole.
is_forest()	Tests if the graph is a forest, i.e. a disjoint union of trees.
is_half_transitive()	Check if self is a half-transitive graph.
is_inscribable()	Test whether the graph is the graph of an inscribed polyhedron.
is_line_graph()	Tests wether the graph is a line graph.
is_long_antihole_free()	Tests whether the given graph contains an induced subgraph that is isomorphic to the complement of a cycle of length at least 5.
is_long_hole_free()	Tests whether g contains an induced cycle of length at least 5.
is_odd_hole_free()	Tests whether self contains an induced odd hole.
is_overfull()	Tests whether the current graph is overfull.
is_partial_cube()	Test whether the given graph is a partial cube.
is_permutation()	Tests whether the graph is a permutation graph.
is_polyhedral()	Check whether the graph is the graph of the polyhedron.
is_prime()	Test whether the current graph is prime.
is_semi_symmetric()	Check if self is semi-symmetric.
is_split()	Returns True if the graph is a Split graph, False otherwise.
is_strongly_regular()	Check whether the graph is strongly regular.
is_tree()	Tests if the graph is a tree
is_triangle_free()	Check whether self is triangle-free
is_weakly_chordal()	Tests whether the given graph is weakly chordal, i.e., the graph and its complement have no induced cycle of length at least 5.

Leftovers

arboricity()	Return the arboricity of the graph and an optional certificate.
common_neighbors_matrix()	Return a matrix of numbers of common neighbors between each pairs.
cores()	Return the core number for each vertex in an ordered list.
effective_resistance()	Return the effective resistance between nodes i and j.
effective_resistance_matrix()	Return a matrix whose (i, j) entry gives the effective resistance between vertices i and j.
fractional_chromatic_index()	Return the fractional chromatic index of the graph.
has_perfect_matching()	Return whether this graph has a perfect matching.
ihara_zeta_function_inverse()	Compute the inverse of the Ihara zeta function of the graph.
kirchhoff_symanzik_polynomial()	Return the Kirchhoff-Symanzik polynomial of a graph.
least_effective_resistance()	Return a list of pairs of nodes with the least effective resistance.
lovasz_theta()	Return the value of Lovász theta-function of graph
magnitude_function()	Return the magnitude function of the graph as a rational function.
matching()	Return a maximum weighted matching of the graph represented by the list of its edges.
maximum_average_degree()	Return the Maximum Average Degree (MAD) of the current graph.
modular_decomposition()	Return the modular decomposition of the current graph.
most_common_neighbors()	Return vertex pairs with maximal number of common neighbors.
perfect_matchings()	Return an iterator over all perfect matchings of the graph.
seidel_adjacency_matrix()	Return the Seidel adjacency matrix of `self`.
seidel_switching()	Return the Seidel switching of `self` w.r.t. subset of vertices `s`.
two_factor_petersen()	Return a decomposition of the graph into 2-factors.
twograph()	Return the two-graph of `self`

Traversals

lex_M()	Return an ordering of the vertices according the LexM graph traversal.
maximum_cardinality_search()	Return an ordering of the vertices according a maximum cardinality search.
maximum_cardinality_search_M()	Return the ordering and the edges of the triangulation produced by MCS-M.

AUTHORS:

- Robert L. Miller (2006-10-22): initial version
- William Stein (2006-12-05): Editing
- **Robert L. Miller (2007-01-13): refactoring, adjusting for NetworkX-0.33, fixed** plotting bugs (2007-01-23): basic tutorial, edge labels, loops, multiple edges and arcs (2007-02-07): graph6 and sparse6 formats, matrix input
- Emily Kirkmann (2007-02-11): added graph_border option to plot and show
- **Robert L. Miller (2007-02-12): vertex color-maps, graph boundaries, graph6** helper functions in Cython
- Robert L. Miller Sage Days 3 (2007-02-17-21): 3d plotting in Tachyon
- Robert L. Miller (2007-02-25): display a partition
- **Robert L. Miller (2007-02-28): associate arbitrary objects to vertices, edge** and arc label display (in 2d), edge coloring
- **Robert L. Miller (2007-03-21): Automorphism group, isomorphism check,** canonical label
- Robert L. Miller (2007-06-07-09): NetworkX function wrapping
- Michael W. Hansen (2007-06-09): Topological sort generation
- Emily Kirkman, Robert L. Miller Sage Days 4: Finished wrapping NetworkX

- **Emily Kirkman (2007-07-21): Genus (including circular planar, all embeddings** and all planar embeddings), all paths, interior paths
- **Bobby Moretti (2007-08-12): fixed up plotting of graphs with edge colors** differentiated by label
- Jason Grout (2007-09-25): Added functions, bug fixes, and general enhancements
- Robert L. Miller (Sage Days 7): Edge labeled graph isomorphism
- Tom Boothby (Sage Days 7): Miscellaneous awesomeness
- Tom Boothby (2008-01-09): Added graphviz output
- David Joyner (2009-2): Fixed docstring bug related to GAP.
- **Stephen Hartke (2009-07-26): Fixed bug in blocks_and_cut_vertices() that** caused an incorrect result when the vertex 0 was a cut vertex.
- **Stephen Hartke (2009-08-22): Fixed bug in blocks_and_cut_vertices() where the** list of cut_vertices is not treated as a set.
- Anders Jonsson (2009-10-10): Counting of spanning trees and out-trees added.
- **Nathann Cohen (2009-09)** [Cliquer, Connectivity, Flows and everything that] uses Linear Programming and class numerical.MIP
- Nicolas M. Thiery (2010-02): graph layout code refactoring, dot2tex/graphviz interface
- David Coudert (2012-04) : Reduction rules in vertex_cover.
- **Birk Eisermann (2012-06): added recognition of weakly chordal graphs and** long-hole-free / long-antihole-free graphs
- Alexandre P. Zuge (2013-07): added join operation.
- Amritanshu Prasad (2014-08): added clique polynomial
- Julian Rüth (2018-06-21): upgrade to NetworkX 2
- David Coudert (2018-10-07): cleaning
- Amanda Francis, Caitlin Lienkaemper, Kate Collins, Rajat Mittal (2019-03-10): methods for computing effective resistance
- Amanda Francis, Caitlin Lienkaemper, Kate Collins, Rajat Mittal (2019-03-19): most_common_neighbors and common_neighbors_matrix added.
- **Jean-Florent Raymond (2019-04): is_redundant, is_dominating,** private_neighbors

1.2.1 Graph Format

Supported formats

Sage Graphs can be created from a wide range of inputs. A few examples are covered here.

- NetworkX dictionary format:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], \
      5: [7, 8], 6: [8,9], 7: [9]}
sage: G = Graph(d); G
Graph on 10 vertices
sage: G.plot().show()     # or G.show()
```

- A NetworkX graph:

```
sage: import networkx
sage: K = networkx.complete_bipartite_graph(12,7)
sage: G = Graph(K)
sage: G.degree()
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12, 12, 12, 12, 12]
```

- graph6 or sparse6 format:

```
sage: s = ':I`AKGsaOs`cI]Gb~'
sage: G = Graph(s, sparse=True); G
Looped multi-graph on 10 vertices
sage: G.plot().show()    # or G.show()
```

Note that the \ character is an escape character in Python, and also a character used by graph6 strings:

```
sage: G = Graph('Ihe\n@GUA')
Traceback (most recent call last):
...
RuntimeError: the string (Ihe) seems corrupt: for n = 10, the string is␣
↪too short
```

In Python, the escaped character \ is represented by \\:

```
sage: G = Graph('Ihe\\n@GUA')
sage: G.plot().show()    # or G.show()
```

- **adjacency matrix: In an adjacency matrix, each column and each row represent a** vertex. If a 1 shows up in row i, column j, there is an edge (i, j).

```
sage: M = Matrix([(0,1,0,0,1,1,0,0,0,0),(1,0,1,0,0,0,1,0,0,0), \
(0,1,0,1,0,0,0,1,0,0), (0,0,1,0,1,0,0,0,1,0),(1,0,0,1,0,0,0,0,0,1), \
(1,0,0,0,0,0,0,1,1,0), (0,1,0,0,0,0,0,0,1,1),(0,0,1,0,0,1,0,0,0,1), \
(0,0,0,1,0,1,1,0,0,0), (0,0,0,0,1,0,1,1,0,0)])
sage: M
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
sage: G = Graph(M); G
Graph on 10 vertices
sage: G.plot().show()    # or G.show()
```

- **incidence matrix: In an incidence matrix, each row represents a vertex and** each column represents an edge.

```
sage: M = Matrix([(-1,  0,  0,  0,  1,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0),
....:             ( 1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0),
....:             ( 0,  1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0),
....:             ( 0,  0,  1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0),
....:             ( 0,  0,  0,  1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1),
....:             ( 0,  0,  0,  0,  0, -1,  0,  0,  0,  1,  1,  0,  0,  0,  0),
```

(continues on next page)

(continued from previous page)

```
....:              ( 0,  0,  0,  0,  0,  0,  0, 1,-1,  0,  0, 1,  0,  0, 0),
....:              ( 0,  0,  0,  0,  0, 1,-1,  0,  0,  0,  0,  0, 1,  0, 0),
....:              ( 0,  0,  0,  0,  0,  0,  0,  0, 1,-1,  0,  0,  0, 1,  0),
....:              ( 0,  0,  0,  0,  0,  0, 1,-1,  0,  0,  0,  0,  0,  0, 1)])
sage: M
[-1  0  0  0  1  0  0  0  0  0 -1  0  0  0  0]
[ 1 -1  0  0  0  0  0  0  0  0  0 -1  0  0  0]
[ 0  1 -1  0  0  0  0  0  0  0  0  0 -1  0  0]
[ 0  0  1 -1  0  0  0  0  0  0  0  0  0 -1  0]
[ 0  0  0  1 -1  0  0  0  0  0  0  0  0  0 -1]
[ 0  0  0  0  0 -1  0  0  0  1  1  0  0  0  0]
[ 0  0  0  0  0  0  0  1 -1  0  0  1  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  1 -1  0  0  0  1  0]
[ 0  0  0  0  0  0  1 -1  0  0  0  0  0  0  1]
sage: G = Graph(M); G
Graph on 10 vertices
sage: G.plot().show()     # or G.show()
sage: DiGraph(matrix(2,[0,0,-1,1]), format="incidence_matrix")
Traceback (most recent call last):
...
ValueError: there must be two nonzero entries (-1 & 1) per column
```

- a list of edges:

```
sage: g = Graph([(1,3),(3,8),(5,2)])
sage: g
Graph on 5 vertices
```

- an igraph Graph:

```
sage: import igraph                                        # optional - python_igraph
sage: g = Graph(igraph.Graph([(1,3),(3,2),(0,2)]))  # optional - python_igraph
sage: g                                                    # optional - python_igraph
Graph on 4 vertices
```

1.2.2 Generators

Use `graphs(n)` to iterate through all non-isomorphic graphs of given size:

```
sage: for g in graphs(4):
....:     print(g.degree_sequence())
[0, 0, 0, 0]
[1, 1, 0, 0]
[2, 1, 1, 0]
[3, 1, 1, 1]
[1, 1, 1, 1]
[2, 2, 1, 1]
[2, 2, 2, 0]
[3, 2, 2, 1]
[2, 2, 2, 2]
[3, 3, 2, 2]
[3, 3, 3, 3]
```

Similarly `graphs()` will iterate through all graphs. The complete graph of 4 vertices is of course the smallest graph with chromatic number bigger than three:

```
sage: for g in graphs():
....:     if g.chromatic_number() > 3:
....:         break
sage: g.is_isomorphic(graphs.CompleteGraph(4))
True
```

For some commonly used graphs to play with, type:

```
sage: graphs.[tab]          # not tested
```

and hit {tab}. Most of these graphs come with their own custom plot, so you can see how people usually visualize these graphs.

```
sage: G = graphs.PetersenGraph()
sage: G.plot().show()     # or G.show()
sage: G.degree_histogram()
[0, 0, 0, 10]
sage: G.adjacency_matrix()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
```

```
sage: S = G.subgraph([0,1,2,3])
sage: S.plot().show()     # or S.show()
sage: S.density()
1/2
```

```
sage: G = GraphQuery(display_cols=['graph6'], num_vertices=7, diameter=5)
sage: L = G.get_graphs_list()
sage: graphs_list.show_graphs(L)
```

1.2.3 Labels

Each vertex can have any hashable object as a label. These are things like strings, numbers, and tuples. Each edge is given a default label of `None`, but if specified, edges can have any label at all. Edges between vertices u and v are represented typically as `(u, v, l)`, where `l` is the label for the edge.

Note that vertex labels themselves cannot be mutable items:

```
sage: M = Matrix( [[0,0],[0,0]] )
sage: G = Graph({ 0 : { M : None } })
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
```

However, if one wants to define a dictionary, with the same keys and arbitrary objects for entries, one can make that association:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), \
      2 : graphs.MoebiusKantorGraph(), 3 : graphs.PetersenGraph() }
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

1.2.4 Database

There is a database available for searching for graphs that satisfy a certain set of parameters, including number of vertices and edges, density, maximum and minimum degree, diameter, radius, and connectivity. To see a list of all search parameter keywords broken down by their designated table names, type

```
sage: graph_db_info()
{...}
```

For more details on data types or keyword input, enter

```
sage: GraphQuery?    # not tested
```

The results of a query can be viewed with the show method, or can be viewed individually by iterating through the results

```
sage: Q = GraphQuery(display_cols=['graph6'],num_vertices=7, diameter=5)
sage: Q.show()
Graph6
--------------------
F?`po
F?gqg
F@?]O
F@OKg
F@R@o
FA_pW
FEOhW
FGC{o
FIAHo
```

Show each graph as you iterate through the results:

```
sage: for g in Q:
....:     show(g)
```

1.2.5 Visualization

To see a graph G you are working with, there are three main options. You can view the graph in two dimensions via matplotlib with `show()`.

```
sage: G = graphs.RandomGNP(15,.3)
sage: G.show()
```

And you can view it in three dimensions via jmol with `show3d()`.

```
sage: G.show3d()
```

Or it can be rendered with LaTeX. This requires the right additions to a standard TeX installation. Then standard Sage commands, such as `view(G)` will display the graph, or `latex(G)` will produce a string suitable for inclusion in a LaTeX document. More details on this are at the *sage.graphs.graph_latex* module.

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: latex(G)
\begin{tikzpicture}
...
\end{tikzpicture}
```

1.2.6 Mutability

Graphs are mutable, and thus unusable as dictionary keys, unless `data_structure="static_sparse"` is used:

```
sage: G = graphs.PetersenGraph()
sage: {G:1}[G]
Traceback (most recent call last):
...
TypeError: This graph is mutable, and thus not hashable. Create an immutable copy by␣
↪`g.copy(immutable=True)`
sage: G_immutable = Graph(G, immutable=True)
sage: G_immutable == G
True
sage: {G_immutable:1}[G_immutable]
1
```

1.2.7 Methods

class sage.graphs.graph.**Graph**(*data=None*, *pos=None*, *loops=None*, *format=None*, *weighted=None*, *data_structure='sparse'*, *vertex_labels=True*, *name=None*, *multiedges=None*, *convert_empty_dict_labels_to_None=None*, *sparse=True*, *immutable=False*)

Bases: *sage.graphs.generic_graph.GenericGraph*

Undirected graph.

A graph is a set of vertices connected by edges. See the Wikipedia article Graph_(mathematics) for more information. For a collection of pre-defined graphs, see the *graph_generators* module.

A *Graph* object has many methods whose list can be obtained by typing `g.<tab>` (i.e. hit the 'tab' key) or by reading the documentation of *graph*, *generic_graph*, and *digraph*.

INPUT:

By default, a *Graph* object is simple (i.e. no *loops* nor *multiple edges*) and unweighted. This can be easily tuned with the appropriate flags (see below).

- `data` – can be any of the following (see the `format` argument):
  1. `Graph()` – build a graph on 0 vertices.
  2. `Graph(5)` – return an edgeless graph on the 5 vertices 0,...,4.

3. `Graph([list_of_vertices, list_of_edges])` – returns a graph with given vertices/edges.

   To bypass auto-detection, prefer the more explicit `Graph([V, E], format='vertices_and_edges')`.

4. `Graph(list_of_edges)` – return a graph with a given list of edges (see documentation of *add_edges()*).

   To bypass auto-detection, prefer the more explicit `Graph(L, format='list_of_edges')`.

5. `Graph({1: [2, 3, 4], 3: [4]})` – return a graph by associating to each vertex the list of its neighbors.

   To bypass auto-detection, prefer the more explicit `Graph(D, format='dict_of_lists')`.

6. `Graph({1: {2: 'a', 3:'b'} ,3:{2:'c'}})` – return a graph by associating a list of neighbors to each vertex and providing its edge label.

   To bypass auto-detection, prefer the more explicit `Graph(D, format='dict_of_dicts')`.

   For graphs with multiple edges, you can provide a list of labels instead, e.g.: `Graph({1: {2: ['a1', 'a2'], 3:['b']} ,3:{2:['c']}})`.

7. `Graph(a_symmetric_matrix)` – return a graph with given (weighted) adjacency matrix (see documentation of *adjacency_matrix()*).

   To bypass auto-detection, prefer the more explicit `Graph(M, format='adjacency_matrix')`. To take weights into account, use `format='weighted_adjacency_matrix'` instead.

8. `Graph(a_nonsymmetric_matrix)` – return a graph with given incidence matrix (see documentation of *incidence_matrix()*).

   To bypass auto-detection, prefer the more explicit `Graph(M, format='incidence_matrix')`.

9. `Graph([V, f])` – return a graph from a vertex set `V` and a *symmetric* function `f`. The graph contains an edge u, v whenever `f(u,v)` is `True`.. Example: `Graph([ [1..10], lambda x, y: abs(x-y).is_square()])`

10. `Graph(':I`ES@obGkqegW~')` – return a graph from a graph6 or sparse6 string (see documentation of *graph6_string()* or *sparse6_string()*).

11. `Graph(a_seidel_matrix, format='seidel_adjacency_matrix')` – return a graph with a given Seidel adjacency matrix (see documentation of *seidel_adjacency_matrix()*).

12. `Graph(another_graph)` – return a graph from a Sage (di)graph, pygraphviz graph, NetworkX graph, or igraph graph.

- `pos` – a positioning dictionary (cf. documentation of *layout()*). For example, to draw 4 vertices on a square:

```
{0: [-1,-1],
 1: [ 1,-1],
 2: [ 1, 1],
 3: [-1, 1]}
```

- **`name` – (must be an explicitly named parameter, i.e.,** `name="complete"`) gives the graph a name
- **`loops` – boolean (default: `None`); whether to allow loops (ignored** if data is an instance of the `Graph` class)

- **multiedges – boolean (default: None); whether to allow multiple** edges (ignored if data is an instance of the Graph class).
- weighted – boolean (default: None); whether graph thinks of itself as weighted or not. See *weighted()*.
- format – if set to None (default), *Graph* tries to guess input's format. To avoid this possibly time-consuming step, one of the following values can be specified (see description above): "int", "graph6", "sparse6", "rule", "list_of_edges", "dict_of_lists", "dict_of_dicts", "adjacency_matrix", "weighted_adjacency_matrix", "seidel_adjacency_matrix", "incidence_matrix", "NX", "igraph".
- sparse – boolean (default: True); sparse=True is an alias for data_structure="sparse", and sparse=False is an alias for data_structure="dense".
- data_structure – one of the following (for more information, see *overview*)
  - "dense" – selects the *dense_graph* backend.
  - "sparse" – selects the *sparse_graph* backend.
  - "static_sparse" – selects the *static_sparse_backend* (this backend is faster than the sparse backend and smaller in memory, and it is immutable, so that the resulting graphs can be used as dictionary keys).
- immutable – boolean (default: False); whether to create a immutable graph. Note that immutable=True is actually a shortcut for data_structure='static_sparse'. Set to False by default.
- vertex_labels – boolean (default: True); whether to allow any object as a vertex (slower), or only the integers $0, ..., n-1$, where n is the number of vertices.
- **convert_empty_dict_labels_to_None – this arguments sets the default** edge labels used by NetworkX (empty dictionaries) to be replaced by None, the default Sage edge label. It is set to True iff a NetworkX graph is on the input.

EXAMPLES:

We illustrate the first seven input formats (the other two involve packages that are currently not standard in Sage):

1. An integer giving the number of vertices:

```
sage: g = Graph(5); g
Graph on 5 vertices
sage: g.vertices()
[0, 1, 2, 3, 4]
sage: g.edges()
[]
```

2. A dictionary of dictionaries:

```
sage: g = Graph({0:{1:'x',2:'z',3:'a'}, 2:{5:'out'}}); g
Graph on 5 vertices
```

The labels ('x', 'z', 'a', 'out') are labels for edges. For example, 'out' is the label for the edge on 2 and 5. Labels can be used as weights, if all the labels share some common parent.:

```
sage: a,b,c,d,e,f = sorted(SymmetricGroup(3))
sage: Graph({b:{d:'c',e:'p'}, c:{d:'p',e:'c'}})
Graph on 4 vertices
```

3. A dictionary of lists:

```
sage: g = Graph({0:[1,2,3], 2:[4]}); g
Graph on 5 vertices
```

4. A list of vertices and a function describing adjacencies. Note that the list of vertices and the function must be enclosed in a list (i.e., [list of vertices, function]).

   Construct the Paley graph over GF(13).:

```
sage: g=Graph([GF(13), lambda i,j: i!=j and (i-j).is_square()])
sage: g.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: g.adjacency_matrix()
[0 1 0 1 1 0 0 0 0 1 1 0 1]
[1 0 1 0 1 1 0 0 0 0 1 1 0]
[0 1 0 1 0 1 1 0 0 0 0 1 1]
[1 0 1 0 1 0 1 1 0 0 0 0 1]
[1 1 0 1 0 1 0 1 1 0 0 0 0]
[0 1 1 0 1 0 1 0 1 1 0 0 0]
[0 0 1 1 0 1 0 1 0 1 1 0 0]
[0 0 0 1 1 0 1 0 1 0 1 1 0]
[0 0 0 0 1 1 0 1 0 1 0 1 1]
[1 0 0 0 0 1 1 0 1 0 1 0 1]
[1 1 0 0 0 0 1 1 0 1 0 1 0]
[0 1 1 0 0 0 0 1 1 0 1 0 1]
[1 0 1 1 0 0 0 0 1 1 0 1 0]
```

   Construct the line graph of a complete graph.:

```
sage: g=graphs.CompleteGraph(4)
sage: line_graph=Graph([g.edges(labels=false), \
       lambda i,j: len(set(i).intersection(set(j)))>0], \
       loops=False)
sage: line_graph.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: line_graph.adjacency_matrix()
[0 1 1 1 1 0]
[1 0 1 1 0 1]
[1 1 0 0 1 1]
[1 1 0 0 1 1]
[1 0 1 1 0 1]
[0 1 1 1 1 0]
```

5. A graph6 or sparse6 string: Sage automatically recognizes whether a string is in graph6 or sparse6 format:

```
sage: s = ':I`AKGsaOs`cI]Gb~'
sage: Graph(s,sparse=True)
Looped multi-graph on 10 vertices
```

```
sage: G = Graph('G?????')
sage: G = Graph("G'?G?C")
Traceback (most recent call last):
...
RuntimeError: the string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sage: G = Graph('G??????')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
RuntimeError: the string (G??????) seems corrupt: for n = 8, the string is␣
↪too long
```

```
sage: G = Graph(":I'AKGsaOs`cI]Gb~")
Traceback (most recent call last):
...
RuntimeError: the string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

There are also list functions to take care of lists of graphs:

```
sage: s = ':IgMoqoCUOqeb\n:I`AKGsaOs`cI]Gb~\n:I`EDOAEQ?PccSsge\\N\n'
sage: graphs_list.from_sparse6(s)
[Looped multi-graph on 10 vertices, Looped multi-graph on 10 vertices, Looped␣
↪multi-graph on 10 vertices]
```

6. A Sage matrix: Note: If format is not specified, then Sage assumes a symmetric square matrix is an adjacency matrix, otherwise an incidence matrix.

   - an adjacency matrix:

```
sage: M = graphs.PetersenGraph().am(); M
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
sage: Graph(M)
Graph on 10 vertices
```

```
sage: Graph(matrix([[1,2],[2,4]]),loops=True,sparse=True)
Looped multi-graph on 2 vertices

sage: M = Matrix([[0,1,-1],[1,0,-1/2],[-1,-1/2,0]]); M
[   0    1   -1]
[   1    0 -1/2]
[  -1 -1/2    0]
sage: G = Graph(M,sparse=True); G
Graph on 3 vertices
sage: G.weighted()
True
```

   - an incidence matrix:

```
sage: M = Matrix(6, [-1,0,0,0,1, 1,-1,0,0,0, 0,1,-1,0,0, 0,0,1,-1,0, 0,0,
↪0,1,-1, 0,0,0,0,0]); M
[-1  0  0  0  1]
[ 1 -1  0  0  0]
[ 0  1 -1  0  0]
[ 0  0  1 -1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  1 -1]
[ 0  0  0  0  0]
sage: Graph(M)
Graph on 6 vertices

sage: Graph(Matrix([[1],[1],[1]]))
Traceback (most recent call last):
...
ValueError: there must be one or two nonzero entries per column in an␣
↪incidence matrix, got entries [1, 1, 1] in column 0
sage: Graph(Matrix([[1],[1],[0]]))
Graph on 3 vertices

sage: M = Matrix([[0,1,-1],[1,0,-1],[-1,-1,0]]); M
[ 0  1 -1]
[ 1  0 -1]
[-1 -1  0]
sage: Graph(M,sparse=True)
Graph on 3 vertices

sage: M = Matrix([[0,1,1],[1,0,1],[-1,-1,0]]); M
[ 0  1  1]
[ 1  0  1]
[-1 -1  0]
sage: Graph(M)
Traceback (most recent call last):
...
ValueError: there must be one or two nonzero entries per column in an␣
↪incidence matrix, got entries [1, 1] in column 2
```

Check that trac ticket #9714 is fixed:

```
sage: MA = Matrix([[1,2,0], [0,2,0], [0,0,1]])
sage: GA = Graph(MA, format='adjacency_matrix')
sage: MI = GA.incidence_matrix(oriented=False)
sage: MI
[2 1 1 0 0 0]
[0 1 1 2 2 0]
[0 0 0 0 0 2]
sage: Graph(MI).edges(labels=None)
[(0, 0), (0, 1), (0, 1), (1, 1), (1, 1), (2, 2)]

sage: M = Matrix([[1], [-1]]); M
[ 1]
[-1]
sage: Graph(M).edges()
[(0, 1, None)]
```

7. A Seidel adjacency matrix:

```
sage: from sage.combinat.matrices.hadamard_matrix import \
....:  regular_symmetric_hadamard_matrix_with_constant_diagonal as rshcd
sage: m=rshcd(16,1)- matrix.identity(16)
sage: Graph(m,format="seidel_adjacency_matrix").is_strongly_
↪regular(parameters=True)
(16, 6, 2, 2)
```

8. List of edges, or labelled edges:

```
sage: g = Graph([(1,3),(3,8),(5,2)])
sage: g
Graph on 5 vertices

sage: g = Graph([(1,2,"Peace"),(7,-9,"and"),(77,2, "Love")])
sage: g
Graph on 5 vertices
sage: g = Graph([(0, 2, '0'), (0, 2, '1'), (3, 3, '2')], loops=True,␣
↪multiedges=True)
sage: g.loops()
[(3, 3, '2')]
```

9. A NetworkX MultiGraph:

```
sage: import networkx
sage: g = networkx.MultiGraph({0:[1,2,3], 2:[4]})
sage: Graph(g)
Graph on 5 vertices
```

10. A NetworkX graph:

```
sage: import networkx
sage: g = networkx.Graph({0:[1,2,3], 2:[4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

11. An igraph Graph (see also *igraph_graph()*):

```
sage: import igraph                          # optional - python_igraph
sage: g = igraph.Graph([(0, 1), (0, 2)]) # optional - python_igraph
sage: Graph(g)                               # optional - python_igraph
Graph on 3 vertices
```

If `vertex_labels` is `True`, the names of the vertices are given by the vertex attribute `'name'`, if available:

```
sage: g = igraph.Graph([(0,1),(0,2)], vertex_attrs={'name':['a','b','c']})  #␣
↪optional - python_igraph
sage: Graph(g).vertices()                                                    #␣
↪optional - python_igraph
['a', 'b', 'c']
sage: g = igraph.Graph([(0,1),(0,2)], vertex_attrs={'label':['a','b','c']}) #␣
↪optional - python_igraph
sage: Graph(g).vertices()                                                    #␣
↪optional - python_igraph
[0, 1, 2]
```

If the igraph Graph has edge attributes, they are used as edge labels:

```
sage: g = igraph.Graph([(0,1),(0,2)], edge_attrs={'name':['a','b'], 'weight
↪':[1,3]}) # optional - python_igraph
sage: Graph(g).edges()                                                       ␣
↪       # optional - python_igraph
[(0, 1, {'name': 'a', 'weight': 1}), (0, 2, {'name': 'b', 'weight': 3})]
```

When defining an undirected graph from a function `f`, it is *very* important that `f` be symmetric. If it is not, anything can happen:

```
sage: f_sym = lambda x,y: abs(x-y) == 1
sage: f_nonsym = lambda x,y: (x-y) == 1
sage: G_sym = Graph([[4,6,1,5,3,7,2,0], f_sym])
sage: G_sym.is_isomorphic(graphs.PathGraph(8))
True
sage: G_nonsym = Graph([[4,6,1,5,3,7,2,0], f_nonsym])
sage: G_nonsym.size()
4
sage: G_nonsym.is_isomorphic(G_sym)
False
```

By default, graphs are mutable and can thus not be used as a dictionary key:

```
sage: G = graphs.PetersenGraph()
sage: {G:1}[G]
Traceback (most recent call last):
...
TypeError: This graph is mutable, and thus not hashable. Create an immutable copy␣
↪by `g.copy(immutable=True)`
```

When providing the optional arguments `data_structure="static_sparse"` or `immutable=True` (both mean the same), then an immutable graph results.

```
sage: G_imm = Graph(G, immutable=True)
sage: H_imm = Graph(G, data_structure='static_sparse')
sage: G_imm == H_imm == G
True
sage: {G_imm:1}[H_imm]
1
```

all_cliques(*graph*, *min_size=0*, *max_size=0*)

Iterator over the cliques in `graph`.

A clique is an induced complete subgraph. This method is an iterator over all the cliques with size in between `min_size` and `max_size`. By default, this method returns only maximum cliques. Each yielded clique is represented by a list of vertices.

Note: Currently only implemented for undirected graphs. Use *to_undirected()* to convert a digraph to an undirected graph.

INPUT:

- `min_size` – integer (default: 0); minimum size of reported cliques. When set to 0 (default), this method searches for maximum cliques. In such case, parameter `max_size` must also be set to 0.
- `max_size` – integer (default: 0); maximum size of reported cliques. When set to 0 (default), the maximum size of the cliques is unbounded. When `min_size` is set to 0, this parameter must be set to 0.

ALGORITHM:

This function is based on Cliquer [NO2003].

EXAMPLES:

```
sage: G = graphs.CompleteGraph(5)
sage: list(sage.graphs.cliquer.all_cliques(G))
```

(continues on next page)

(continued from previous page)

```
[[0, 1, 2, 3, 4]]
sage: list(sage.graphs.cliquer.all_cliques(G, 2, 3))
[[3, 4],
 [2, 3],
 [2, 3, 4],
 [2, 4],
 [1, 2],
 [1, 2, 3],
 [1, 2, 4],
 [1, 3],
 [1, 3, 4],
 [1, 4],
 [0, 1],
 [0, 1, 2],
 [0, 1, 3],
 [0, 1, 4],
 [0, 2],
 [0, 2, 3],
 [0, 2, 4],
 [0, 3],
 [0, 3, 4],
 [0, 4]]
sage: G.delete_edge([1,3])
sage: list(sage.graphs.cliquer.all_cliques(G))
[[0, 2, 3, 4], [0, 1, 2, 4]]
```

Todo: Use the re-entrant functionality of Cliquer [NO2003] to avoid storing all cliques.

apex_vertices(*k=None*)

Return the list of apex vertices.

A graph is apex if it can be made planar by the removal of a single vertex. The deleted vertex is called `an apex` of the graph, and a graph may have more than one apex. For instance, in the minimal nonplanar graphs K_5 or $K_{3,3}$, every vertex is an apex. The apex graphs include graphs that are themselves planar, in which case again every vertex is an apex. The null graph is also counted as an apex graph even though it has no vertex to remove. If the graph is not connected, we say that it is apex if it has at most one non planar connected component and that this component is apex. See the Wikipedia article Apex_graph for more information.

See also:

- *is_apex()*
- *is_planar()*

INPUT:

- `k` – integer (default: `None`); when set to `None`, the method returns the list of all apex of the graph, possibly empty if the graph is not apex. When set to a positive integer, the method ends as soon as k apex vertices are found.

OUTPUT:

By default, the method returns the list of all apex of the graph. When parameter `k` is set to a positive integer, the returned list is bounded to k apex vertices.

EXAMPLES:

K_5 and $K_{3,3}$ are apex graphs, and each of their vertices is an apex:

```
sage: G = graphs.CompleteGraph(5)
sage: G.apex_vertices()
[0, 1, 2, 3, 4]
sage: G = graphs.CompleteBipartiteGraph(3,3)
sage: G.is_apex()
True
sage: G.apex_vertices()
[0, 1, 2, 3, 4, 5]
sage: G.apex_vertices(k=3)
[0, 1, 2]
```

A 4
$times$4-grid is apex and each of its vertices is an apex. When adding a universal vertex, the resulting graph is apex and the universal vertex is the unique apex vertex

```
sage: G = graphs.Grid2dGraph(4,4)
sage: set(G.apex_vertices()) == set(G.vertices())
True
sage: G.add_edges([('universal',v) for v in G])
sage: G.apex_vertices()
['universal']
```

The Petersen graph is not apex:

```
sage: G = graphs.PetersenGraph()
sage: G.apex_vertices()
[]
```

A graph is apex if all its connected components are apex, but at most one is not planar:

```
sage: M = graphs.Grid2dGraph(3,3)
sage: K5 = graphs.CompleteGraph(5)
sage: (M+K5).apex_vertices()
[9, 10, 11, 12, 13]
sage: (M+K5+K5).apex_vertices()
[]
```

Neighbors of an apex of degree 2 are apex:

```
sage: G = graphs.Grid2dGraph(5,5)
sage: v = (666, 666)
sage: G.add_path([(1, 1), v, (3, 3)])
sage: G.is_planar()
False
sage: G.degree(v)
2
sage: sorted(G.apex_vertices())
[(1, 1), (2, 2), (3, 3), (666, 666)]
```

arboricity(*certificate=False*)

Return the arboricity of the graph and an optional certificate.

The arboricity is the minimum number of forests that covers the graph.

See Wikipedia article Arboricity

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate.

OUTPUT:

When `certificate = True`, then the function returns (a, F) where a is the arboricity and F is a list of a disjoint forests that partitions the edge set of g. The forests are represented as subgraphs of the original graph.

If `certificate = False`, the function returns just a integer indicating the arboricity.

ALGORITHM:

Represent the graph as a graphical matroid, then apply matroid `sage.matroid.partition()` algorithm from the matroids module.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: a,F = G.arboricity(True)
sage: a
2
sage: all([f.is_forest() for f in F])
True
sage: len(set.union(*[set(f.edges()) for f in F])) == G.size()
True
```

atoms_and_clique_separators(*G*, *tree=False*, *rooted_tree=False*, *separators=False*)

Return the atoms of the decomposition of G by clique minimal separators.

Let $G = (V, E)$ be a graph. A set $S \subset V$ is a clique separator if $G[S]$ is a clique and the graph $G \setminus S$ has at least 2 connected components. Let $C \subset V$ be the vertices of a connected component of $G \setminus S$. The graph $G[C + S]$ is an *atom* if it has no clique separator.

This method implements the algorithm proposed in [BPS2010], that improves upon the algorithm proposed in [TY1984], for computing the atoms and the clique minimal separators of a graph. This algorithm is based on the *maximum_cardinality_search_M()* graph traversal and has time complexity in $O(|V| \cdot |E|)$.

If the graph is not connected, we insert empty separators between the lists of separators of each connected components. See the examples below for more details.

INPUT:

- `G` – a Sage graph
- `tree` – boolean (default: `False`); whether to return the result as a directed tree in which internal nodes are clique separators and leaves are the atoms of the decomposition. Since a clique separator is repeated when its removal partition the graph into 3 or more connected components, vertices are labels by tuples (i, S), where S is the set of vertices of the atom or the clique separator, and $0 \leq i \leq |T|$.
- `rooted_tree` – boolean (default: `False`); whether to return the result as a `LabelledRootedTree`. When `tree` is `True`, this parameter is ignored.
- `separators` – boolean (default: `False`); whether to also return the complete list of separators considered during the execution of the algorithm. When `tree` or `rooted_tree` is `True`, this parameter is ignored.

OUTPUT:

- By default, return a tuple (A, S_c), where A is the list of atoms of the graph in the order of dicovery, and S_c is the list of clique separators, with possible repetitions, in the order the separator has been considered. If furthermore `separators` is `True`, return a tuple (A, S_h, S_c), where S_c is the list of considered separators of the graph in the order they have been considered.

- When `tree` is `True`, format the result as a directed tree
- When `rooted_tree` is `True` and `tree` is `False`, format the ouput as a `LabelledRootedTree`

EXAMPLES:

Example of [BPS2010]:

```
sage: G = Graph({'a': ['b', 'k'], 'b': ['c'], 'c': ['d', 'j', 'k'],
....:            'd': ['e', 'f', 'j', 'k'], 'e': ['g'],
....:            'f': ['g', 'j', 'k'], 'g': ['j', 'k'], 'h': ['i', 'j'],
....:            'i': ['k'], 'j': ['k']})
sage: atoms, cliques = G.atoms_and_clique_separators()
sage: sorted(sorted(a) for a in atoms)
[['a', 'b', 'c', 'k'],
 ['c', 'd', 'j', 'k'],
 ['d', 'e', 'f', 'g', 'j', 'k'],
 ['h', 'i', 'j', 'k']]
sage: sorted(sorted(c) for c in cliques)
[['c', 'k'], ['d', 'j', 'k'], ['j', 'k']]
sage: T = G.atoms_and_clique_separators(tree=True)
sage: T.is_tree()
True
sage: T.diameter() == len(atoms)
True
sage: all(u[1] in atoms for u in T if T.degree(u) == 1)
True
sage: all(u[1] in cliques for u in T if T.degree(u) != 1)
True
```

A graph without clique separator:

```
sage: G = graphs.CompleteGraph(5)
sage: G.atoms_and_clique_separators()
([{0, 1, 2, 3, 4}], [])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
{0, 1, 2, 3, 4}
```

Graphs with several biconnected components:

```
sage: G = graphs.PathGraph(4)
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _____{2}____
 /            /
{2, 3}    __{1}__
         /       /
       {1, 2} {0, 1}

sage: G = graphs.WindmillGraph(3, 4)
sage: G.atoms_and_clique_separators()
([{0, 1, 2}, {0, 3, 4}, {0, 5, 6}, {0, 8, 7}], [{0}, {0}, {0}])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _________{0}_________
 /                     /
{0, 1, 2}    ________{0}_______
            /                  /
          {0, 3, 4}    _____{0}___
```

(continues on next page)

(continued from previous page)

```
                      /          /
                    {0, 8, 7} {0, 5, 6}
```

When the removal of a clique separator results in $k > 2$ connected components, this separator is repeated $k - 1$ times, but the repetitions are not necessarily contiguous:

```
sage: G = Graph(2)
sage: for i in range(5):
....:     G.add_cycle([0, 1, G.add_vertex()])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  __________{0, 1}_____
 /                    /
{0, 1, 4}   _________{0, 1}_____
           /                    /
          {0, 1, 2}   ________{0, 1}___
                     /                 /
                    {0, 1, 3}   _____{0, 1}
                               /           /
                              {0, 1, 5} {0, 1, 6}

sage: G = graphs.StarGraph(3)
sage: G.subdivide_edges(G.edges(), 2)
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{5}_______
 /                /
{1, 5}   _______{7}_______
        /                /
       {2, 7}   _______{9}_______
               /                 /
              {9, 3}   _______{6}_______
                      /                 /
                     {6, 7}   _______{4}_____
                             /               /
                            {4, 5}   ______{0}_____
                                    /              /
                                   {0, 6}   _____{8}____
                                           /           /
                                          {8, 9}   __{0}__
                                                  /       /
                                                 {0, 8} {0, 4}
```

If the graph is not connected, we insert empty separators between the lists of separators of each connected components. For instance, let G be a graph with 3 connected components. The method returns the list $S_c = [S_0, \cdots, S_i, \ldots, S_j, \ldots, S_{k-1}]$ of k clique separators, where i and j are the indexes of the inserted empty separators and $0 \leq i < j < k - 1$. The method also returns the list $A = [A_0, \ldots, S_k]$ of the $k + 1$ atoms, with $k + 1 \geq 3$. The lists of atoms and clique separators of each of the connected components are respectively $[A_0, \ldots, A_i]$ and $[S_0, \ldots, S_{i-1}]$, $[A_{i+1}, \ldots, A_j]$ and $[S_{i+1}, \ldots, S_{j-1}]$, and $[A_{j+1}, \ldots, A_k]$ and $[S_{j+1}, \ldots, S_{k-1}]$. One can check that for each connected component, we get one atom more than clique separators:

```
sage: G = graphs.PathGraph(3) * 3
sage: A, Sc = G.atoms_and_clique_separators()
sage: A
[{1, 2}, {0, 1}, {4, 5}, {3, 4}, {8, 7}, {6, 7}]
sage: Sc
[{1}, {}, {4}, {}, {7}]
```

(continues on next page)

(continued from previous page)

```
sage: i , j = [i for i, s in enumerate(Sc) if not s]
sage: i, j
(1, 3)
sage: A[:i+1], Sc[:i]
([{1, 2}, {0, 1}], [{1}])
sage: A[i+1:j+1], Sc[i+1:j]
([{4, 5}, {3, 4}], [{4}])
sage: A[j+1:], Sc[j+1:]
([{8, 7}, {6, 7}], [{7}])
sage: I = [-1, i, j, len(Sc)]
sage: for i, j in zip(I[:-1], I[1:]):
....:     print(A[i+1:j+1], Sc[i+1:j])
[{1, 2}, {0, 1}] [{1}]
[{4, 5}, {3, 4}] [{4}]
[{8, 7}, {6, 7}] [{7}]
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{1}_______
 /                 /
{1, 2}    _______{}_______
         /               /
        {0, 1}    ______{4}______
                 /              /
                {4, 5}    _____{}_____
                         /            /
                        {3, 4}    __{7}__
                                 /       /
                                {6, 7} {8, 7}
```

Loops and multiple edges are ignored:

```
sage: G.allow_loops(True)
sage: G.add_edges([(u, u) for u in G])
sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edges())
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{1}_______
 /                 /
{1, 2}    _______{}_______
         /               /
        {0, 1}    ______{4}______
                 /              /
                {4, 5}    _____{}_____
                         /            /
                        {3, 4}    __{7}__
                                 /       /
                                {6, 7} {8, 7}
```

We can check that the returned list of separators is valid:

```
sage: G = graphs.RandomGNP(50, .1)
sage: while not G.is_connected():
....:     G = graphs.RandomGNP(50, .1)
sage: _, separators, _ = G.atoms_and_clique_separators(separators=True)
sage: for S in separators:
....:     H = G.copy()
....:     H.delete_vertices(S)
....:     if H.is_connected():
```

(continues on next page)

(continued from previous page)

```
....:         raise ValueError("something goes wrong")
```

bipartite_color()

Return a dictionary with vertices as the keys and the color class as the values.

Fails with an error if the graph is not bipartite.

EXAMPLES:

```
sage: graphs.CycleGraph(4).bipartite_color()
{0: 1, 1: 0, 2: 1, 3: 0}
sage: graphs.CycleGraph(5).bipartite_color()
Traceback (most recent call last):
...
RuntimeError: Graph is not bipartite.
```

bipartite_sets()

Return (X, Y) where X and Y are the nodes in each bipartite set of graph G.

Fails with an error if graph is not bipartite.

EXAMPLES:

```
sage: graphs.CycleGraph(4).bipartite_sets()
({0, 2}, {1, 3})
sage: graphs.CycleGraph(5).bipartite_sets()
Traceback (most recent call last):
...
RuntimeError: Graph is not bipartite.
```

bounded_outdegree_orientation(*bound*, *solver=None*, *verbose=False*)

Computes an orientation of `self` such that every vertex v has out-degree less than $b(v)$

INPUT:

- `bound` – Maximum bound on the out-degree. Can be of three different types :
- An integer k. In this case, computes an orientation whose maximum out-degree is less than k.
- A dictionary associating to each vertex its associated maximum out-degree.
- A function associating to each vertex its associated maximum out-degree.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

A DiGraph representing the orientation if it exists. A `ValueError` exception is raised otherwise.

ALGORITHM:

The problem is solved through a maximum flow :

Given a graph G, we create a `DiGraph` D defined on $E(G) \cup V(G) \cup \{s, t\}$. We then link s to all of $V(G)$ (these edges having a capacity equal to the bound associated to each element of $V(G)$), and all the elements of $E(G)$ to t . We then link each $v \in V(G)$ to each of its incident edges in G. A maximum integer flow of value $|E(G)|$ corresponds to an admissible orientation of G. Otherwise, none exists.

EXAMPLES:

There is always an orientation of a graph G such that a vertex v has out-degree at most $\lceil \frac{d(v)}{2} \rceil$:

```
sage: g = graphs.RandomGNP(40, .4)
sage: b = lambda v: ceil(g.degree(v)/2)
sage: D = g.bounded_outdegree_orientation(b)
sage: all( D.out_degree(v) <= b(v) for v in g )
True
```

Chvatal's graph, being 4-regular, can be oriented in such a way that its maximum out-degree is 2:

```
sage: g = graphs.ChvatalGraph()
sage: D = g.bounded_outdegree_orientation(2)
sage: max(D.out_degree())
2
```

For any graph G, it is possible to compute an orientation such that the maximum out-degree is at most the maximum average degree of G divided by 2. Anything less, though, is impossible.

sage: g = graphs.RandomGNP(40, .4) sage: mad = g.maximum_average_degree()

Hence this is possible

```
sage: d = g.bounded_outdegree_orientation(ceil(mad/2))
```

While this is not:

```
sage: try:
....:     g.bounded_outdegree_orientation(ceil(mad/2-1))
....:     print("Error")
....: except ValueError:
....:     pass
```

bridges(*G*, *labels=True*)

Return a list of the bridges (or cut edges).

A bridge is an edge whose deletion disconnects the undirected graph. A disconnected graph has no bridge.

INPUT:

- `labels` – boolean (default: `True`); if `False`, each bridge is a tuple (u, v) of vertices

EXAMPLES:

```
sage: from sage.graphs.connectivity import bridges
sage: from sage.graphs.connectivity import is_connected
sage: g = 2 * graphs.PetersenGraph()
sage: g.add_edge(1, 10)
sage: is_connected(g)
True
sage: bridges(g)
[(1, 10, None)]
sage: g.bridges()
[(1, 10, None)]
```

centrality_degree(*v=None*)

Return the degree centrality of a vertex.

The degree centrality of a vertex v is its degree, divided by $|V(G)| - 1$. For more information, see the Wikipedia article Centrality.

INPUT:

- v – a vertex (default: None); set to None (default) to get a dictionary associating each vertex with its centrality degree.

See also:

- *centrality_closeness()*
- *centrality_betweenness()*

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_degree()
{0: 4/11, 1: 4/11, 2: 4/11, 3: 4/11,  4: 4/11,  5: 4/11,
 6: 4/11, 7: 4/11, 8: 4/11, 9: 4/11, 10: 4/11, 11: 4/11}
sage: D = graphs.DiamondGraph()
sage: D.centrality_degree()
{0: 2/3, 1: 1, 2: 1, 3: 2/3}
sage: D.centrality_degree(v=1)
1
```

chromatic_index(*solver=None*, *verbose=0*)

Return the chromatic index of the graph.

The chromatic index is the minimal number of colors needed to properly color the edges of the graph.

INPUT:

- solver – (default: None); specify the Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.

This method is a frontend for method *sage.graphs.graph_coloring.edge_coloring()* that uses a mixed integer-linear programming formulation to compute the chromatic index.

See also:

- Wikipedia article Edge_coloring for further details on edge coloring
- *sage.graphs.graph_coloring.edge_coloring()*
- *fractional_chromatic_index()*
- *chromatic_number()*

EXAMPLES:

The clique K_n has chromatic index n when n is odd and $n-1$ when n is even:

```
sage: graphs.CompleteGraph(4).chromatic_index()
3
sage: graphs.CompleteGraph(5).chromatic_index()
5
sage: graphs.CompleteGraph(6).chromatic_index()
5
```

The path P_n with $n \geq 2$ has chromatic index 2:

```
sage: graphs.PathGraph(5).chromatic_index()
2
```

The windmill graph with parameters k, n has chromatic index $(k-1)n$:

```
sage: k,n = 3,4
sage: G = graphs.WindmillGraph(k,n)
sage: G.chromatic_index() == (k-1)*n
True
```

chromatic_number(*algorithm='DLX'*, *solver=None*, *verbose=0*)

Return the minimal number of colors needed to color the vertices of the graph.

INPUT:

- `algorithm` – Select an algorithm from the following supported algorithms:
  - If `algorithm="DLX"` (default), the chromatic number is computed using the dancing link algorithm. It is inefficient speedwise to compute the chromatic number through the dancing link algorithm because this algorithm computes *all* the possible colorings to check that one exists.
  - If `algorithm="CP"`, the chromatic number is computed using the coefficients of the chromatic polynomial. Again, this method is inefficient in terms of speed and it only useful for small graphs.
  - If `algorithm="MILP"`, the chromatic number is computed using a mixed integer linear program. The performance of this implementation is affected by whether optional MILP solvers have been installed (see the `MILP module`, or Sage's tutorial on Linear Programming).
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity for the MILP algorithm. Its default value is 0, which means *quiet*.

See also:

For more functions related to graph coloring, see the module *sage.graphs.graph_coloring*.

EXAMPLES:

```
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: G.chromatic_number(algorithm="DLX")
3
sage: G.chromatic_number(algorithm="MILP")
3
sage: G.chromatic_number(algorithm="CP")
3
```

A bipartite graph has (by definition) chromatic number 2:

```
sage: graphs.RandomBipartite(50,50,0.7).chromatic_number()
2
```

A complete multipartite graph with k parts has chromatic number k:

```
sage: all(graphs.CompleteMultipartiteGraph([5]*i).chromatic_number() == i for␣
↪i in range(2,5))
True
```

The complete graph has the largest chromatic number from all the graphs of order n. Namely its chromatic number is n:

```
sage: all(graphs.CompleteGraph(i).chromatic_number() == i for i in range(10))
True
```

The Kneser graph with parameters $(n, 2)$ for $n > 3$ has chromatic number $n - 2$:

```
sage: all(graphs.KneserGraph(i,2).chromatic_number() == i-2 for i in range(4,
↪6))
True
```

The Flower Snark graph has chromatic index 4 hence its line graph has chromatic number 4:

```
sage: graphs.FlowerSnark().line_graph().chromatic_number()
4
```

chromatic_polynomial(*G*, *return_tree_basis=False*)

Compute the chromatic polynomial of the graph G.

The algorithm used is a recursive one, based on the following observations of Read:

- The chromatic polynomial of a tree on n vertices is x(x-1)^(n-1).
- If e is an edge of G, G' is the result of deleting the edge e, and G" is the result of contracting e, then the chromatic polynomial of G is equal to that of G' minus that of G'‘.

EXAMPLES:

```
sage: graphs.CycleGraph(4).chromatic_polynomial()
x^4 - 4*x^3 + 6*x^2 - 3*x
sage: graphs.CycleGraph(3).chromatic_polynomial()
x^3 - 3*x^2 + 2*x
sage: graphs.CubeGraph(3).chromatic_polynomial()
x^8 - 12*x^7 + 66*x^6 - 214*x^5 + 441*x^4 - 572*x^3 + 423*x^2 - 133*x
sage: graphs.PetersenGraph().chromatic_polynomial()
x^10 - 15*x^9 + 105*x^8 - 455*x^7 + 1353*x^6 - 2861*x^5 + 4275*x^4 - 4305*x^3␣
↪+ 2606*x^2 - 704*x
sage: graphs.CompleteBipartiteGraph(3,3).chromatic_polynomial()
x^6 - 9*x^5 + 36*x^4 - 75*x^3 + 78*x^2 - 31*x
sage: for i in range(2,7):
....:     graphs.CompleteGraph(i).chromatic_polynomial().factor()
(x - 1) * x
(x - 2) * (x - 1) * x
(x - 3) * (x - 2) * (x - 1) * x
(x - 4) * (x - 3) * (x - 2) * (x - 1) * x
(x - 5) * (x - 4) * (x - 3) * (x - 2) * (x - 1) * x
sage: graphs.CycleGraph(5).chromatic_polynomial().factor()
(x - 2) * (x - 1) * x * (x^2 - 2*x + 2)
sage: graphs.OctahedralGraph().chromatic_polynomial().factor()
(x - 2) * (x - 1) * x * (x^3 - 9*x^2 + 29*x - 32)
sage: graphs.WheelGraph(5).chromatic_polynomial().factor()
(x - 2) * (x - 1) * x * (x^2 - 5*x + 7)
sage: graphs.WheelGraph(6).chromatic_polynomial().factor()
(x - 3) * (x - 2) * (x - 1) * x * (x^2 - 4*x + 5)
sage: C(x)=graphs.LCFGraph(24, [12,7,-7], 8).chromatic_polynomial()  # long␣
↪time (6s on sage.math, 2011)
sage: C(2)  # long time
0
```

By definition, the chromatic number of a graph G is the least integer k such that the chromatic polynomial of G is strictly positive at k:

```
sage: G = graphs.PetersenGraph()
sage: P = G.chromatic_polynomial()
sage: min(i for i in range(11) if P(i) > 0) == G.chromatic_number()
True

sage: G = graphs.RandomGNP(10,0.7)
sage: P = G.chromatic_polynomial()
sage: min(i for i in range(11) if P(i) > 0) == G.chromatic_number()
True
```

chromatic_quasisymmetric_function(*t=None*, *R=None*)

Return the chromatic quasisymmetric function of `self`.

Let G be a graph whose vertex set is totally ordered. The chromatic quasisymmetric function $X_G(t)$ was first described in [SW2012]. We use the equivalent definition given in [BC2018]:

$$X_G(t) = \sum_{\sigma=(\sigma_1,\ldots,\sigma_n)} t^{\mathrm{asc}(\sigma)} M_{|\sigma_1|,\ldots,|\sigma_n|},$$

where we sum over all ordered set partitions of the vertex set of G such that each block σ_i is an independent (i.e., stable) set of G, and where $\mathrm{asc}(\sigma)$ denotes the number of edges $\{u, v\}$ of G such that $u < v$ and v appears in a later part of σ than u.

INPUT:

- `t` – (optional) the parameter t; uses the variable t in $\mathbf{Z}[t]$ by default
- `R` – (optional) the base ring for the quasisymmetric functions; uses the parent of t by default

EXAMPLES:

```
sage: G = Graph([[1,2,3], [[1,3], [2,3]]])
sage: G.chromatic_quasisymmetric_function()
(2*t^2+2*t+2)*M[1, 1, 1] + M[1, 2] + t^2*M[2, 1]
sage: G = graphs.PathGraph(4)
sage: XG = G.chromatic_quasisymmetric_function(); XG
(t^3+11*t^2+11*t+1)*M[1, 1, 1, 1] + (3*t^2+3*t)*M[1, 1, 2]
 + (3*t^2+3*t)*M[1, 2, 1] + (3*t^2+3*t)*M[2, 1, 1]
 + (t^2+t)*M[2, 2]
sage: XG.to_symmetric_function()
(t^3+11*t^2+11*t+1)*m[1, 1, 1, 1] + (3*t^2+3*t)*m[2, 1, 1]
 + (t^2+t)*m[2, 2]
sage: G = graphs.CompleteGraph(4)
sage: G.chromatic_quasisymmetric_function()
(t^6+3*t^5+5*t^4+6*t^3+5*t^2+3*t+1)*M[1, 1, 1, 1]
```

Not all chromatic quasisymmetric functions are symmetric:

```
sage: G = Graph([[1,2], [1,5], [3,4], [3,5]])
sage: G.chromatic_quasisymmetric_function().is_symmetric()
False
```

We check that at $t = 1$, we recover the usual chromatic symmetric function:

```
sage: p = SymmetricFunctions(QQ).p()
sage: G = graphs.CycleGraph(5)
```

(continues on next page)

(continued from previous page)

```
sage: XG = G.chromatic_quasisymmetric_function(t=1); XG
120*M[1, 1, 1, 1, 1] + 30*M[1, 1, 1, 2] + 30*M[1, 1, 2, 1]
 + 30*M[1, 2, 1, 1] + 10*M[1, 2, 2] + 30*M[2, 1, 1, 1]
 + 10*M[2, 1, 2] + 10*M[2, 2, 1]
sage: p(XG.to_symmetric_function())
p[1, 1, 1, 1, 1] - 5*p[2, 1, 1, 1] + 5*p[2, 2, 1]
 + 5*p[3, 1, 1] - 5*p[3, 2] - 5*p[4, 1] + 4*p[5]

sage: G = graphs.ClawGraph()
sage: XG = G.chromatic_quasisymmetric_function(t=1); XG
24*M[1, 1, 1, 1] + 6*M[1, 1, 2] + 6*M[1, 2, 1] + M[1, 3]
 + 6*M[2, 1, 1] + M[3, 1]
sage: p(XG.to_symmetric_function())
p[1, 1, 1, 1] - 3*p[2, 1, 1] + 3*p[3, 1] - p[4]
```

chromatic_symmetric_function(*R=None*)

Return the chromatic symmetric function of `self`.

Let G be a graph. The chromatic symmetric function X_G was described in [Sta1995], specifically Theorem 2.5 states that

$$X_G = \sum_{F \subseteq E(G)} (-1)^{|F|} p_{\lambda(F)},$$

where $\lambda(F)$ is the partition of the sizes of the connected components of the subgraph induced by the edges F and p_μ is the powersum symmetric function.

INPUT:

- `R` – (optional) the base ring for the symmetric functions; this uses $\mathbf{Z}$ by default

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: G = graphs.CycleGraph(5)
sage: XG = G.chromatic_symmetric_function(); XG
p[1, 1, 1, 1, 1] - 5*p[2, 1, 1, 1] + 5*p[2, 2, 1]
 + 5*p[3, 1, 1] - 5*p[3, 2] - 5*p[4, 1] + 4*p[5]
sage: s(XG)
30*s[1, 1, 1, 1, 1] + 10*s[2, 1, 1, 1] + 10*s[2, 2, 1]
```

Not all graphs have a positive Schur expansion:

```
sage: G = graphs.ClawGraph()
sage: XG = G.chromatic_symmetric_function(); XG
p[1, 1, 1, 1] - 3*p[2, 1, 1] + 3*p[3, 1] - p[4]
sage: s(XG)
8*s[1, 1, 1, 1] + 5*s[2, 1, 1] - s[2, 2] + s[3, 1]
```

We show that given a triangle $\{e_1, e_2, e_3\}$, we have $X_G = X_{G-e_1} + X_{G-e_2} - X_{G-e_1-e_2}$:

```
sage: G = Graph([[1,2],[1,3],[2,3]])
sage: XG = G.chromatic_symmetric_function()
sage: G1 = copy(G)
sage: G1.delete_edge([1,2])
sage: XG1 = G1.chromatic_symmetric_function()
sage: G2 = copy(G)
sage: G2.delete_edge([1,3])
```

(continues on next page)

(continued from previous page)

```
sage: XG2 = G2.chromatic_symmetric_function()
sage: G3 = copy(G1)
sage: G3.delete_edge([1,3])
sage: XG3 = G3.chromatic_symmetric_function()
sage: XG == XG1 + XG2 - XG3
True
```

cleave(*G*, *cut_vertices=None*, *virtual_edges=True*, *solver=None*, *verbose=0*)

Return the connected subgraphs separated by the input vertex cut.

Given a connected (multi)graph G and a vertex cut X, this method computes the list of subgraphs of G induced by each connected component c of $G \setminus X$ plus X, i.e., $G[c \cup X]$.

INPUT:

- `G` – a Graph.
- `cut_vertices` – iterable container of vertices (default: `None`); a set of vertices representing a vertex cut of `G`. If no vertex cut is given, the method will compute one via a call to *vertex_connectivity()*.
- `virtual_edges` – boolean (default: `True`); whether to add virtual edges to the sides of the cut or not. A virtual edge is an edge between a pair of vertices of the cut that are not connected by an edge in `G`.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `sage.numerical.mip.MixedIntegerLinearProgram.solve()` of the class `sage.numerical.mip.MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT: A triple (S, C, f), where

- S is a list of the graphs that are sides of the vertex cut.
- C is the graph of the cocycles. For each pair of vertices of the cut, if there exists an edge between them, C has one copy of each edge connecting them in `G` per sides of the cut plus one extra copy. Furthermore, when `virtual_edges == True`, if a pair of vertices of the cut is not connected by an edge in `G`, then it has one virtual edge between them per sides of the cut.
- f is the complement of the subgraph of `G` induced by the vertex cut. Hence, its vertex set is the vertex cut, and its edge set is the set of virtual edges (i.e., edges between pairs of vertices of the cut that are not connected by an edge in `G`). When `virtual_edges == False`, the edge set is empty.

EXAMPLES:

If there is an edge between cut vertices:

```
sage: from sage.graphs.connectivity import cleave
sage: G = Graph(2)
sage: for _ in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: S1,C1,f1 = cleave(G, cut_vertices=[0, 1])
sage: [g.order() for g in S1]
[4, 4, 4]
sage: C1.order(), C1.size()
(2, 4)
sage: f1.vertices(), f1.edges()
([0, 1], [])
```

If `virtual_edges == False` and there is an edge between cut vertices:

```
sage: G.subgraph([0, 1]).complement() == Graph([[0, 1], []])
True
sage: S2,C2,f2 = cleave(G, cut_vertices=[0, 1], virtual_edges=False)
sage: (S1 == S2, C1 == C2, f1 == f2)
(True, True, True)
```

If cut vertices doesn't have edge between them:

```
sage: G.delete_edge(0, 1)
sage: S1,C1,f1 = cleave(G, cut_vertices=[0, 1])
sage: [g.order() for g in S1]
[4, 4, 4]
sage: C1.order(), C1.size()
(2, 3)
sage: f1.vertices(), f1.edges()
([0, 1], [(0, 1, None)])
```

If `virtual_edges == False` and the cut vertices are not connected by an edge:

```
sage: G.subgraph([0, 1]).complement() == Graph([[0, 1], []])
False
sage: S2,C2,f2 = cleave(G, cut_vertices=[0, 1], virtual_edges=False)
sage: [g.order() for g in S2]
[4, 4, 4]
sage: C2.order(), C2.size()
(2, 0)
sage: f2.vertices(), f2.edges()
([0, 1], [])
sage: (S1 == S2, C1 == C2, f1 == f2)
(False, False, False)
```

If G is a biconnected multigraph:

```
sage: G = graphs.CompleteBipartiteGraph(2, 3)
sage: G.add_edge(2, 3)
sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edge_iterator())
sage: G.add_edges([(0, 1), (0, 1), (0, 1)])
sage: S,C,f = cleave(G, cut_vertices=[0, 1])
sage: for g in S:
....:     print(g.edges(labels=0))
[(0, 1), (0, 1), (0, 1), (0, 2), (0, 2), (0, 3), (0, 3), (1, 2), (1, 2), (1,␣
↪3), (1, 3), (2, 3), (2, 3)]
[(0, 1), (0, 1), (0, 1), (0, 4), (0, 4), (1, 4), (1, 4)]
```

clique_complex()

Return the clique complex of self.

This is the largest simplicial complex on the vertices of self whose 1-skeleton is self.

This is only makes sense for undirected simple graphs.

EXAMPLES:

```
sage: g = Graph({0:[1,2],1:[2],4:[]})
sage: g.clique_complex()
Simplicial complex with vertex set (0, 1, 2, 4) and facets {(4,), (0, 1, 2)}
```

(continues on next page)

(continued from previous page)

```
sage: h = Graph({0:[1,2,3,4],1:[2,3,4],2:[3]})
sage: x = h.clique_complex()
sage: x
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(0, 1, 4), (0,␣
↪1, 2, 3)}
sage: i = x.graph()
sage: i==h
True
sage: x==i.clique_complex()
True
```

clique_maximum(*algorithm='Cliquer'*, *solver=None*, *verbose=0*)

Return the vertex set of a maximal order complete subgraph.

INPUT:

- `algorithm` – the algorithm to be used :
  - If `algorithm = "Cliquer"` (default), wraps the C program Cliquer [NO2003].
  - If `algorithm = "MILP"`, the problem is solved through a Mixed Integer Linear Program.

    (see `MixedIntegerLinearProgram`)
  - If `algorithm = "mcqd"`, uses the MCQD solver (http://www.sicmm.org/~konc/maxclique/). Note that the MCQD package must be installed.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Parameters `solver` and `verbose` are used only when `algorithm="MILP"`.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

ALGORITHM:

This function is based on Cliquer [NO2003].

EXAMPLES:

Using Cliquer (default):

```
sage: C = graphs.PetersenGraph()
sage: C.clique_maximum()
[7, 9]
sage: C = Graph('DJ{')
sage: C.clique_maximum()
[1, 2, 3, 4]
```

Through a Linear Program:

```
sage: len(C.clique_maximum(algorithm="MILP"))
4
```

clique_number(*algorithm='Cliquer'*, *cliques=None*, *solver=None*, *verbose=0*)

Return the order of the largest clique of the graph

This is also called as the clique number.

Note: Currently only implemented for undirected graphs. Use `to_undirected` to convert a digraph to an undirected graph.

INPUT:

- `algorithm` – the algorithm to be used :
  - If `algorithm = "Cliquer"`, wraps the C program Cliquer [NO2003].
  - If `algorithm = "networkx"`, uses the NetworkX's implementation of the Bron and Kerbosch Algorithm [BK1973].
  - If `algorithm = "MILP"`, the problem is solved through a Mixed Integer Linear Program.

    (see `MixedIntegerLinearProgram`)
  - If `algorithm = "mcqd"`, uses the MCQD solver (http://www.sicmm.org/~konc/maxclique/). Note that the MCQD package must be installed.
- `cliques` – an optional list of cliques that can be input if already computed. Ignored unless `algorithm=="networkx"`.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

ALGORITHM:

This function is based on Cliquer [NO2003] and [BK1973].

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.clique_number()
4
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.clique_number()
3
```

By definition the clique number of a complete graph is its order:

```
sage: all(graphs.CompleteGraph(i).clique_number() == i for i in range(1,15))
True
```

A non-empty graph without edges has a clique number of 1:

```
sage: all((i*graphs.CompleteGraph(1)).clique_number() == 1 for i in range(1,
↪15))
True
```

A complete multipartite graph with k parts has clique number k:

```
sage: all((i*graphs.CompleteMultipartiteGraph(i*[5])).clique_number() == i␣
↪for i in range(1,6))
True
```

clique_polynomial(*t=None*)

Return the clique polynomial of self.

This is the polynomial where the coefficient of t^n is the number of cliques in the graph with n vertices. The constant term of the clique polynomial is always taken to be one.

EXAMPLES:

```
sage: g = Graph()
sage: g.clique_polynomial()
1
sage: g = Graph({0:[1]})
sage: g.clique_polynomial()
t^2 + 2*t + 1
sage: g = graphs.CycleGraph(4)
sage: g.clique_polynomial()
4*t^2 + 4*t + 1
```

cliques_containing_vertex(*vertices=None*, *cliques=None*)

Return the cliques containing each vertex, represented as a dictionary of lists of lists, keyed by vertex.

Returns a single list if only one input vertex.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `vertices` – the vertices to inspect (default is entire graph)
- `cliques` – list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_containing_vertex()
{0: [[4, 0]], 1: [[4, 1, 2, 3]], 2: [[4, 1, 2, 3]], 3: [[4, 1, 2, 3]], 4: [[4,
↪ 0], [4, 1, 2, 3]]}
sage: E = C.cliques_maximal()
sage: E
[[0, 4], [1, 2, 3, 4]]
sage: C.cliques_containing_vertex(cliques=E)
{0: [[0, 4]], 1: [[1, 2, 3, 4]], 2: [[1, 2, 3, 4]], 3: [[1, 2, 3, 4]], 4: [[0,
↪ 4], [1, 2, 3, 4]]}

sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_containing_vertex()
{0: [[0, 1, 2], [0, 1, 3]], 1: [[0, 1, 2], [0, 1, 3]], 2: [[0, 1, 2]], 3: [[0,
↪ 1, 3]]}
```

Since each clique of a 2 dimensional grid corresponds to an edge, the number of cliques in which a vertex is involved equals its degree:

```
sage: F = graphs.Grid2dGraph(2,3)
sage: d = F.cliques_containing_vertex()
sage: all(F.degree(u) == len(cliques) for u,cliques in d.items())
True
sage: F.cliques_containing_vertex(vertices=[(0, 1)])
{(0, 1): [[(0, 1), (0, 0)], [(0, 1), (0, 2)], [(0, 1), (1, 1)]]}
```

cliques_get_clique_bipartite(***kwds*)

Return a bipartite graph constructed such that maximal cliques are the right vertices and the left vertices are retained from the given graph. Right and left vertices are connected if the bottom vertex belongs to the clique represented by a top vertex.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).cliques_get_clique_bipartite()
Bipartite graph on 36 vertices
sage: ((graphs.ChvatalGraph()).cliques_get_clique_bipartite()).
↪show(figsize=[2,2], vertex_size=20, vertex_labels=False)
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_get_clique_bipartite()
Bipartite graph on 6 vertices
sage: (G.cliques_get_clique_bipartite()).show(figsize=[2,2])
```

cliques_get_max_clique_graph()

Return the clique graph.

Vertices of the result are the maximal cliques of the graph, and edges of the result are between maximal cliques with common members in the original graph.

For more information, see the Wikipedia article Clique_graph.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).cliques_get_max_clique_graph()
Graph on 24 vertices
sage: ((graphs.ChvatalGraph()).cliques_get_max_clique_graph()).
↪show(figsize=[2,2], vertex_size=20, vertex_labels=False)
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_get_max_clique_graph()
Graph on 2 vertices
sage: (G.cliques_get_max_clique_graph()).show(figsize=[2,2])
```

cliques_maximal(*algorithm='native'*)

Return the list of all maximal cliques.

Each clique is represented by a list of vertices. A clique is an induced complete subgraph, and a maximal clique is one not contained in a larger one.

INPUT:

- `algorithm` – can be set to `"native"` (default) to use Sage's own implementation, or to `"NetworkX"` to use NetworkX' implementation of the Bron and Kerbosch Algorithm [BK1973].

Note: This method sorts its output before returning it. If you prefer to save the extra time, you can call *sage.graphs.independent_sets.IndependentSets* directly.

Note: Sage's implementation of the enumeration of *maximal* independent sets is not much faster than NetworkX' (expect a 2x speedup), which is surprising as it is written in Cython. This being said, the algorithm from NetworkX appears to be sligthly different from this one, and that would be a good thing to explore if one wants to improve the implementation.

ALGORITHM:

This function is based on NetworkX's implementation of the Bron and Kerbosch Algorithm [BK1973].

EXAMPLES:

```
sage: graphs.ChvatalGraph().cliques_maximal()
[[0, 1], [0, 4], [0, 6], [0, 9], [1, 2], [1, 5], [1, 7], [2, 3],
 [2, 6], [2, 8], [3, 4], [3, 7], [3, 9], [4, 5], [4, 8], [5, 10],
 [5, 11], [6, 10], [6, 11], [7, 8], [7, 11], [8, 10], [9, 10], [9, 11]]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2, 2])
sage: G.cliques_maximal()
[[0, 1, 2], [0, 1, 3]]
sage: C = graphs.PetersenGraph()
sage: C.cliques_maximal()
[[0, 1], [0, 4], [0, 5], [1, 2], [1, 6], [2, 3], [2, 7], [3, 4],
 [3, 8], [4, 9], [5, 7], [5, 8], [6, 8], [6, 9], [7, 9]]
sage: C = Graph('DJ{')
sage: C.cliques_maximal()
[[0, 4], [1, 2, 3, 4]]
```

Comparing the two implementations:

```
sage: g = graphs.RandomGNP(20,.7)
sage: s1 = Set(map(Set, g.cliques_maximal(algorithm="NetworkX")))
sage: s2 = Set(map(Set, g.cliques_maximal(algorithm="native")))
sage: s1 == s2
True
```

cliques_maximum(*graph*)

Returns the vertex sets of *ALL* the maximum complete subgraphs.

Returns the list of all maximum cliques, with each clique represented by a list of vertices. A clique is an induced complete subgraph, and a maximum clique is one of maximal order.

Note: Currently only implemented for undirected graphs. Use *to_undirected()* to convert a digraph to an undirected graph.

ALGORITHM:

This function is based on Cliquer [NO2003].

EXAMPLES:

```
sage: graphs.ChvatalGraph().cliques_maximum() # indirect doctest
[[0, 1], [0, 4], [0, 6], [0, 9], [1, 2], [1, 5], [1, 7], [2, 3],
 [2, 6], [2, 8], [3, 4], [3, 7], [3, 9], [4, 5], [4, 8], [5, 10],
 [5, 11], [6, 10], [6, 11], [7, 8], [7, 11], [8, 10], [9, 10], [9, 11]]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_maximum()
[[0, 1, 2], [0, 1, 3]]
sage: C = graphs.PetersenGraph()
sage: C.cliques_maximum()
[[0, 1], [0, 4], [0, 5], [1, 2], [1, 6], [2, 3], [2, 7], [3, 4],
 [3, 8], [4, 9], [5, 7], [5, 8], [6, 8], [6, 9], [7, 9]]
sage: C = Graph('DJ{')
sage: C.cliques_maximum()
[[1, 2, 3, 4]]
```

cliques_number_of(*vertices=None*, *cliques=None*)

Return a dictionary of the number of maximal cliques containing each vertex, keyed by vertex.

This returns a single value if only one input vertex.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `vertices` – the vertices to inspect (default is entire graph)
- `cliques` – list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_number_of()
{0: 1, 1: 1, 2: 1, 3: 1, 4: 2}
sage: E = C.cliques_maximal()
sage: E
[[0, 4], [1, 2, 3, 4]]
sage: C.cliques_number_of(cliques=E)
{0: 1, 1: 1, 2: 1, 3: 1, 4: 2}
sage: F = graphs.Grid2dGraph(2,3)
sage: F.cliques_number_of()
{(0, 0): 2, (0, 1): 3, (0, 2): 2, (1, 0): 2, (1, 1): 3, (1, 2): 2}
sage: F.cliques_number_of(vertices=[(0, 1), (1, 2)])
{(0, 1): 3, (1, 2): 2}
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_number_of()
{0: 2, 1: 2, 2: 1, 3: 1}
```

cliques_vertex_clique_number(*algorithm='cliquer'*, *vertices=None*, *cliques=None*)

Return a dictionary of sizes of the largest maximal cliques containing each vertex, keyed by vertex.

Returns a single value if only one input vertex.

Note: Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an

undirected graph.

INPUT:

- `algorithm` – either `cliquer` or `networkx`
  - `cliquer` – This wraps the C program Cliquer [NO2003].
  - `networkx` – This function is based on NetworkX's implementation of the Bron and Kerbosch Algorithm [BK1973].
- `vertices` – the vertices to inspect (default is entire graph). Ignored unless `algorithm=='networkx'`.
- `cliques` – list of cliques (if already computed). Ignored unless `algorithm=='networkx'`.

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_vertex_clique_number()
{0: 2, 1: 4, 2: 4, 3: 4, 4: 4}
sage: E = C.cliques_maximal()
sage: E
[[0, 4], [1, 2, 3, 4]]
sage: C.cliques_vertex_clique_number(cliques=E,algorithm="networkx")
{0: 2, 1: 4, 2: 4, 3: 4, 4: 4}
sage: F = graphs.Grid2dGraph(2,3)
sage: F.cliques_vertex_clique_number(algorithm="networkx")
{(0, 0): 2, (0, 1): 2, (0, 2): 2, (1, 0): 2, (1, 1): 2, (1, 2): 2}
sage: F.cliques_vertex_clique_number(vertices=[(0, 1), (1, 2)])
{(0, 1): 2, (1, 2): 2}
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_vertex_clique_number()
{0: 3, 1: 3, 2: 3, 3: 3}
```

coloring(*algorithm='DLX'*, *hex_colors=False*, *solver=None*, *verbose=0*)

Return the first (optimal) proper vertex-coloring found.

INPUT:

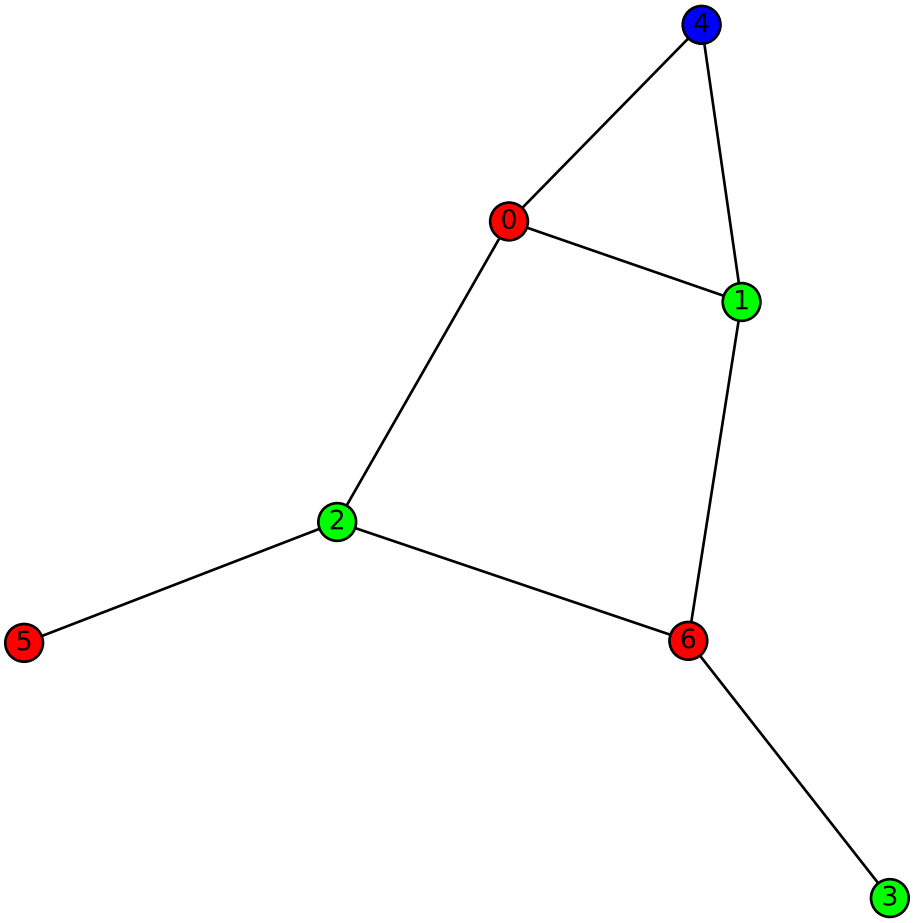
- `algorithm` – Select an algorithm from the following supported algorithms:
  - If `algorithm="DLX"` (default), the coloring is computed using the dancing link algorithm.
  - If `algorithm="MILP"`, the coloring is computed using a mixed integer linear program. The performance of this implementation is affected by whether optional MILP solvers have been installed (see the `MILP module`).
- `hex_colors` – boolean (default: `False`); if `True`, return a dictionary which can easily be used for plotting.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity for the MILP algorithm. Its default value is 0, which means *quiet*.

See also:

For more functions related to graph coloring, see the module *sage.graphs.graph_coloring*.

EXAMPLES:

```
sage: G = Graph("Fooba")
sage: P = G.coloring(algorithm="MILP")
sage: Q = G.coloring(algorithm="DLX")
sage: def are_equal_colorings(A, B):
....:     return Set(map(Set, A)) == Set(map(Set, B))
sage: are_equal_colorings(P, [[1, 2, 3], [0, 5, 6], [4]])
True
sage: are_equal_colorings(P, Q)
True
sage: G.plot(partition=P)
Graphics object consisting of 16 graphics primitives
sage: G.coloring(hex_colors=True, algorithm="MILP")
{'#0000ff': [4], '#00ff00': [0, 6, 5], '#ff0000': [2, 1, 3]}
sage: H = G.coloring(hex_colors=True, algorithm="DLX")
sage: H
{'#0000ff': [4], '#00ff00': [1, 2, 3], '#ff0000': [0, 5, 6]}
sage: G.plot(vertex_colors=H)
Graphics object consisting of 16 graphics primitives
```



common_neighbors_matrix(*vertices=None*, *nonedgesonly=True*)

Return a matrix of numbers of common neighbors between each pairs.

The (i, j) entry of the matrix gives the number of common neighbors between vertices i and j.

This method is only valid for simple (no loops, no multiple edges) graphs.

INPUT:

- `nonedgesonly`– boolean (default: `True`); if `True`, assigns 0 value to adjacent vertices.
- `vertices` – list (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *GenericGraph.vertices()* is used.

OUTPUT: matrix

EXAMPLES:

The common neighbors matrix for a straight linear 2-tree counting only non-adjacent vertex pairs

```
sage: G1 = Graph()
sage: G1.add_edges([(0,1),(0,2),(1,2),(1,3),(3,5),(2,4),(2,3),(3,4),(4,5)])
sage: G1.common_neighbors_matrix(nonedgesonly = True)
[0 0 0 2 1 0]
[0 0 0 0 2 1]
[0 0 0 0 0 2]
[2 0 0 0 0 0]
[1 2 0 0 0 0]
[0 1 2 0 0 0]
```

We now show the common neighbors matrix which includes adjacent vertices

```
sage: G1.common_neighbors_matrix(nonedgesonly = False)
[0 1 1 2 1 0]
[1 0 2 1 2 1]
[1 2 0 2 1 2]
[2 1 2 0 2 1]
[1 2 1 2 0 1]
[0 1 2 1 1 0]
```

The common neighbors matrix for a fan on 6 vertices counting only non-adjacent vertex pairs

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.common_neighbors_matrix()
[0 0 0 0 0 0 0]
[0 0 0 2 1 1 1]
[0 0 0 0 2 1 1]
[0 2 0 0 0 2 1]
[0 1 2 0 0 0 1]
[0 1 1 2 0 0 1]
[0 1 1 1 1 1 0]
```

It is an error to input anything other than a simple graph:

```
sage: G = Graph([(0,0)],loops=True)
sage: G.common_neighbors_matrix()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with loops.
Perhaps this method can be updated to handle them, but in the
meantime if you want to use it please disallow loops using
allow_loops().
```

See also:

- *most_common_neighbors()* – returns node pairs with most shared neighbors

convexity_properties()

Return a `ConvexityProperties` object corresponding to `self`.

This object contains the methods related to convexity in graphs (convex hull, hull number) and caches useful information so that it becomes comparatively cheaper to compute the convex hull of many different sets of the same graph.

See also:

In order to know what can be done through this object, please refer to module *sage.graphs.convexity_properties*.

Note: If you want to compute many convex hulls, keep this object in memory ! When it is created, it builds a table of useful information to compute convex hulls. As a result

```
sage: g = graphs.PetersenGraph()
sage: g.convexity_properties().hull([1, 3])
[1, 2, 3]
sage: g.convexity_properties().hull([3, 7])
[2, 3, 7]
```

Is a terrible waste of computations, while

```
sage: g = graphs.PetersenGraph()
sage: CP = g.convexity_properties()
sage: CP.hull([1, 3])
[1, 2, 3]
sage: CP.hull([3, 7])
[2, 3, 7]
```

Makes perfect sense.

---

cores(*k=None*, *with_labels=False*)

Return the core number for each vertex in an ordered list.

(for homomorphisms cores, see the *Graph.has_homomorphism_to()* method)

DEFINITIONS:

- *K-cores* in graph theory were introduced by Seidman in 1983 and by Bollobas in 1984 as a method of (destructively) simplifying graph topology to aid in analysis and visualization. They have been more recently defined as the following by Batagelj et al:

  Given a graph 'G' with vertices set 'V' and edges set 'E', the 'k'-core of 'G' is the graph obtained from 'G' by recursively removing the vertices with degree less than 'k', for as long as there are any.

  This operation can be useful to filter or to study some properties of the graphs. For instance, when you compute the 2-core of graph G, you are cutting all the vertices which are in a tree part of graph. (A tree is a graph with no loops). See the Wikipedia article K-core.

  [PSW1996] defines a k-core of G as the largest subgraph (it is unique) of G with minimum degree at least k.

- Core number of a vertex

  The core number of a vertex v is the largest integer k such that v belongs to the k-core of G.

- Degeneracy

The *degeneracy* of a graph G, usually denoted $\delta^*(G)$, is the smallest integer k such that the graph G can be reduced to the empty graph by iteratively removing vertices of degree $\leq k$. Equivalently, $\delta^*(G) = k$ if k is the smallest integer such that the k-core of G is empty.

IMPLEMENTATION:

This implementation is based on the NetworkX implementation of the algorithm described in [BZ2003].

INPUT:

- `k` – integer (default: `None`);
  - If `k = None` (default), returns the core number for each vertex.
  - If `k` is an integer, returns a pair `(ordering, core)`, where `core` is the list of vertices in the k-core of `self`, and `ordering` is an elimination order for the other vertices such that each vertex is of degree strictly less than k when it is to be eliminated from the graph.
- `with_labels` – boolean (default: `False`); when set to `False`, and `k = None`, the method returns a list whose i th element is the core number of the i th vertex. When set to `True`, the method returns a dictionary whose keys are vertices, and whose values are the corresponding core numbers.

See also:

- Graph cores is also a notion related to graph homomorphisms. For this second meaning, see *Graph.has_homomorphism_to()*.

EXAMPLES:

```
sage: (graphs.FruchtGraph()).cores()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
sage: (graphs.FruchtGraph()).cores(with_labels=True)
{0: 3, 1: 3, 2: 3, 3: 3, 4: 3, 5: 3, 6: 3, 7: 3, 8: 3, 9: 3, 10: 3, 11: 3}
sage: a = random_matrix(ZZ, 20, x=2, sparse=True, density=.1)
sage: b = Graph(20)
sage: b.add_edges(a.nonzero_positions(), loops=False)
sage: cores = b.cores(with_labels=True); cores
{0: 3, 1: 3, 2: 3, 3: 3, 4: 2, 5: 2, 6: 3, 7: 1, 8: 3, 9: 3, 10: 3, 11: 3,␣
↪12: 3, 13: 3, 14: 2, 15: 3, 16: 3, 17: 3, 18: 3, 19: 3}
sage: [v for v,c in cores.items() if c >= 2] # the vertices in the 2-core
[0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Checking the 2-core of a random lobster is indeed the empty set:

```
sage: g = graphs.RandomLobster(20, .5, .5)
sage: ordering, core = g.cores(2)
sage: len(core) == 0
True
```

degree_constrained_subgraph(*bounds*, *solver=None*, *verbose=0*)

Returns a degree-constrained subgraph.

Given a graph G and two functions $f, g : V(G) \to \mathbb{Z}$ such that $f \leq g$, a degree-constrained subgraph in G is a subgraph $G' \subseteq G$ such that for any vertex $v \in G$, $f(v) \leq d_{G'}(v) \leq g(v)$.

INPUT:

- `bounds` – (default: `None`); Two possibilities:
  - A dictionary whose keys are the vertices, and values a pair of real values `(min,max)` corresponding to the values $(f(v), g(v))$.

  - A function associating to each vertex a pair of real values `(min,max)` corresponding to the values $(f(v), g(v))$.

- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

- When a solution exists, this method outputs the degree-constrained subgraph as a Graph object.
- When no solution exists, returns `False`.

Note:

- This algorithm computes the degree-constrained subgraph of minimum weight.
- If the graph's edges are weighted, these are taken into account.
- This problem can be solved in polynomial time.

EXAMPLES:

Is there a perfect matching in an even cycle?

```
sage: g = graphs.CycleGraph(6)
sage: bounds = lambda x: [1,1]
sage: m = g.degree_constrained_subgraph(bounds=bounds)
sage: m.size()
3
```

ear_decomposition()

Return an Ear decomposition of the graph.

An ear of an undirected graph G is a path P where the two endpoints of the path may coincide (i.e., form a cycle), but where otherwise no repetition of edges or vertices is allowed, so every internal vertex of P has degree two in P.

An ear decomposition of an undirected graph G is a partition of its set of edges into a sequence of ears, such that the one or two endpoints of each ear belong to earlier ears in the sequence and such that the internal vertices of each ear do not belong to any earlier ear.

For more information, see the Wikipedia article Ear_decomposition.

This method implements the linear time algorithm presented in [Sch2013].

OUTPUT:

- A nested list representing the cycles and chains of the ear decomposition of the graph.

EXAMPLES:

Ear decomposition of an outer planar graph of order 13:

```
sage: g = Graph('LlCG{O@?GBOMW?')
sage: g.ear_decomposition()
[[0, 3, 2, 1, 0],
 [0, 7, 4, 3],
 [0, 11, 9, 8, 7],
 [1, 12, 2],
```

(continues on next page)

(continued from previous page)

```
 [3, 6, 5, 4],
 [4, 6],
 [7, 10, 8],
 [7, 11],
 [8, 11]]
```

Ear decomposition of a biconnected graph:

```
sage: g = graphs.CycleGraph(4)
sage: g.ear_decomposition()
[[0, 3, 2, 1, 0]]
```

Ear decomposition of a connected but not biconnected graph:

```
sage: G = Graph()
sage: G.add_cycle([0,1,2])
sage: G.add_edge(0,3)
sage: G.add_cycle([3,4,5,6])
sage: G.ear_decomposition()
[[0, 2, 1, 0], [3, 6, 5, 4, 3]]
```

The ear decomposition of a multigraph with loops is the same as the ear decomposition of the underlying simple graph:

```
sage: g = graphs.BullGraph()
sage: g.allow_multiple_edges(True)
sage: g.add_edges(g.edges())
sage: g.allow_loops(True)
sage: u = g.random_vertex()
sage: g.add_edge(u, u)
sage: g
Bull graph: Looped multi-graph on 5 vertices
sage: h = g.to_simple()
sage: g.ear_decomposition() == h.ear_decomposition()
True
```

effective_resistance(i, j)

Return the effective resistance between nodes i and j.

The resistance distance between vertices i and j of a simple connected graph G is defined as the effective resistance between the two vertices on an electrical network constructed from G replacing each edge of the graph by a unit (1 ohm) resistor.

See the Wikipedia article Resistance_distance for more information.

INPUT:

- `i`, `j` – vertices of the graph

OUTPUT: rational number denoting resistance between nodes i and j

EXAMPLES:

Effective resistances in a straight linear 2-tree on 6 vertices

```
sage: G = Graph([(0,1),(0,2),(1,2),(1,3),(3,5),(2,4),(2,3),(3,4),(4,5)])
sage: G.effective_resistance(0,1)
34/55
sage: G.effective_resistance(0,3)
```

(continues on next page)

(continued from previous page)

```
49/55
sage: G.effective_resistance(1,4)
9/11
sage: G.effective_resistance(0,5)
15/11
```

Effective resistances in a fan on 6 vertices

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.effective_resistance(1,5)
6/5
sage: H.effective_resistance(1,3)
49/55
```

See also:

- *effective_resistance_matrix()* – a similar method giving a matrix full of all effective resistances between all nodes
- *least_effective_resistance()* – gives node pairs with least effective resistances
- See Wikipedia article Resistance_distance for more details.

effective_resistance_matrix(*vertices=None*, *nonedgesonly=True*)

Return a matrix whose (i, j) entry gives the effective resistance between vertices i and j.

The resistance distance between vertices i and j of a simple connected graph G is defined as the effective resistance between the two vertices on an electrical network constructed from G replacing each edge of the graph by a unit (1 ohm) resistor.

INPUT:

- `nonedgesonly` – boolean (default: `True`); if `True` assign zero resistance to pairs of adjacent vertices.
- `vertices` – list (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *GenericGraph.vertices()* is used.

OUTPUT: matrix

EXAMPLES:

The effective resistance matrix for a straight linear 2-tree counting only non-adjacent vertex pairs

```
sage: G = Graph([(0,1),(0,2),(1,2),(1,3),(3,5),(2,4),(2,3),(3,4),(4,5)])
sage: G.effective_resistance_matrix()
[    0     0     0 49/55 59/55 15/11]
[    0     0     0     0  9/11 59/55]
[    0     0     0     0     0 49/55]
[49/55     0     0     0     0     0]
[59/55  9/11     0     0     0     0]
[15/11 59/55 49/55     0     0     0]
```

The same effective resistance matrix, this time including adjacent vertices

```
sage: G.effective_resistance_matrix(nonedgesonly=False)
[    0 34/55 34/55 49/55 59/55 15/11]
[34/55     0 26/55 31/55  9/11 59/55]
[34/55 26/55     0  5/11 31/55 49/55]
```

(continues on next page)

(continued from previous page)

```
[49/55 31/55  5/11     0 26/55 34/55]
[59/55  9/11 31/55 26/55     0 34/55]
[15/11 59/55 49/55 34/55 34/55     0]
```

This example illustrates the common neighbors matrix for a fan on 6 vertices counting only non-adjacent vertex pairs

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.effective_resistance_matrix()
[    0     0     0     0     0     0     0]
[    0     0     0 49/55 56/55   6/5 89/55]
[    0     0     0     0   4/5 56/55 81/55]
[    0 49/55     0     0     0 49/55 16/11]
[    0 56/55   4/5     0     0     0 81/55]
[    0   6/5 56/55 49/55     0     0 89/55]
[    0 89/55 81/55 16/11 81/55 89/55     0]
```

See also:

- *least_effective_resistance()* – gives node pairs with least effective resistances
- *effective_resistance()* – computes effective resistance for a single node pair
- See Wikipedia article Resistance_Distance for more details.

fractional_chromatic_index(*solver='PPL'*, *verbose_constraints=False*, *verbose=0*)

Return the fractional chromatic index of the graph.

The fractional chromatic index is a relaxed version of edge-coloring. An edge coloring of a graph being actually a covering of its edges into the smallest possible number of matchings, the fractional chromatic index of a graph G is the smallest real value $\chi_f(G)$ such that there exists a list of matchings $M_1, ..., M_k$ of G and coefficients $\alpha_1, ..., \alpha_k$ with the property that each edge is covered by the matchings in the following relaxed way

$$\forall e \in E(G), \sum_{e \in M_i} \alpha_i \geq 1$$

For more information, see the Wikipedia article Fractional_coloring.

ALGORITHM:

The fractional chromatic index is computed through Linear Programming through its dual. The LP solved by sage is actually:

$$\text{Maximize : } \sum_{e \in E(G)} r_e$$

$$\text{Such that : } \forall M \text{ matching } \subseteq G, \sum_{e \in M} r_v \leq 1$$

INPUT:

- `solver` – (default: `"PPL"`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.

Note: The default solver used here is `"PPL"` which provides exact results, i.e. a rational number, although this may be slower that using other solvers. Be aware that this method may loop endlessly when using some non exact solvers as reported in trac ticket #23658 and trac ticket #23798.

- `verbose_constraints` – boolean (default: `False`); whether to display which constraints are being generated.
- `verbose` – integer (default: 0); sets the level of verbosity of the LP solver.

EXAMPLES:

The fractional chromatic index of a C_5 is $5/2$:

```
sage: g = graphs.CycleGraph(5)
sage: g.fractional_chromatic_index()
5/2
```

gomory_hu_tree(*algorithm=None*)

Return a Gomory-Hu tree of self.

Given a tree T with labeled edges representing capacities, it is very easy to determine the maximum flow between any pair of vertices : it is the minimal label on the edges of the unique path between them.

Given a graph G, a Gomory-Hu tree T of G is a tree with the same set of vertices, and such that the maximum flow between any two vertices is the same in G as in T. See the Wikipedia article Gomory–Hu_tree. Note that, in general, a graph admits more than one Gomory-Hu tree.

See also 15.4 (Gomory-Hu trees) from [Sch2003].

INPUT:

- `algorithm` – select the algorithm used by the `edge_cut()` method. Refer to its documentation for allowed values and default behaviour.

OUTPUT:

A graph with labeled edges

EXAMPLES:

Taking the Petersen graph:

```
sage: g = graphs.PetersenGraph()
sage: t = g.gomory_hu_tree()
```

Obviously, this graph is a tree:

```
sage: t.is_tree()
True
```

Note that if the original graph is not connected, then the Gomory-Hu tree is in fact a forest:

```
sage: (2*g).gomory_hu_tree().is_forest()
True
sage: (2*g).gomory_hu_tree().is_connected()
False
```

On the other hand, such a tree has lost nothing of the initial graph connectedness:

```
sage: all(t.flow(u,v) == g.flow(u,v) for u,v in Subsets(g.vertices(), 2))
True
```

Just to make sure, we can check that the same is true for two vertices in a random graph:

```
sage: g = graphs.RandomGNP(20,.3)
sage: t = g.gomory_hu_tree()
sage: g.flow(0,1) == t.flow(0,1)
True
```

And also the min cut:

```
sage: g.edge_connectivity() == min(t.edge_labels())
True
```

graph6_string()

Return the graph6 representation of the graph as an ASCII string.

This is only valid for simple (no loops, no multiple edges) graphs on at most $2^{18} - 1 = 262143$ vertices.

Note: As the graph6 format only handles graphs with vertex set $\{0, ..., n-1\}$, a *relabelled copy* will be encoded, if necessary.

See also:

- *dig6_string()* – a similar string format for directed graphs

EXAMPLES:

```
sage: G = graphs.KrackhardtKiteGraph()
sage: G.graph6_string()
'IvUqwK@?G'
```

has_homomorphism_to(*H*, *core=False*, *solver=None*, *verbose=0*)

Checks whether there is a homomorphism between two graphs.

A homomorphism from a graph G to a graph H is a function $\phi : V(G) \mapsto V(H)$ such that for any edge $uv \in E(G)$ the pair $\phi(u)\phi(v)$ is an edge of H.

Saying that a graph can be k-colored is equivalent to saying that it has a homomorphism to K_k, the complete graph on k elements.

For more information, see the Wikipedia article Graph_homomorphism.

INPUT:

- `H` – the graph to which `self` should be sent.
- `core` – boolean (default: `False`; whether to minimize the size of the mapping's image (see note below). This is set to `False` by default.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: One can compute the core of a graph (with respect to homomorphism) with this method

```
sage: g = graphs.CycleGraph(10)
sage: mapping = g.has_homomorphism_to(g, core = True)
sage: print("The size of the core is {}".format(len(set(mapping.values()))))
The size of the core is 2
```

OUTPUT:

This method returns `False` when the homomorphism does not exist, and returns the homomorphism otherwise as a dictionary associating a vertex of H to a vertex of G.

EXAMPLES:

Is Petersen's graph 3-colorable:

```
sage: P = graphs.PetersenGraph()
sage: P.has_homomorphism_to(graphs.CompleteGraph(3)) is not False
True
```

An odd cycle admits a homomorphism to a smaller odd cycle, but not to an even cycle:

```
sage: g = graphs.CycleGraph(9)
sage: g.has_homomorphism_to(graphs.CycleGraph(5)) is not False
True
sage: g.has_homomorphism_to(graphs.CycleGraph(7)) is not False
True
sage: g.has_homomorphism_to(graphs.CycleGraph(4)) is not False
False
```

has_perfect_matching(*algorithm='Edmonds'*, *solver=None*, *verbose=0*)

Return whether this graph has a perfect matching.

INPUT:

- `algorithm` – string (default: `"Edmonds"`)
  - `"Edmonds"` uses Edmonds' algorithm as implemented in NetworkX to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `"LP_matching"` uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `"LP"` uses a Linear Program formulation of the perfect matching problem: put a binary variable `b[e]` on each edge e, and for each vertex v, require that the sum of the values of the edges incident to v is 1.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used; if set to `None`, the default one is used
- `verbose` – integer (default: `0`); sets the level of verbosity: set to 0 by default, which means quiet (only useful when `algorithm == "LP_matching"` or `algorithm == "LP"`)

For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.

OUTPUT:

A boolean.

EXAMPLES:

```
sage: graphs.PetersenGraph().has_perfect_matching()
True
sage: graphs.WheelGraph(6).has_perfect_matching()
True
sage: graphs.WheelGraph(5).has_perfect_matching()
False
sage: graphs.PetersenGraph().has_perfect_matching(algorithm="LP_matching")
True
sage: graphs.WheelGraph(6).has_perfect_matching(algorithm="LP_matching")
True
sage: graphs.WheelGraph(5).has_perfect_matching(algorithm="LP_matching")
False
sage: graphs.PetersenGraph().has_perfect_matching(algorithm="LP_matching")
True
sage: graphs.WheelGraph(6).has_perfect_matching(algorithm="LP_matching")
True
sage: graphs.WheelGraph(5).has_perfect_matching(algorithm="LP_matching")
False
```

ihara_zeta_function_inverse()

Compute the inverse of the Ihara zeta function of the graph.

This is a polynomial in one variable with integer coefficients. The Ihara zeta function itself is the inverse of this polynomial.

See the Wikipedia article Ihara zeta function for more information.

ALGORITHM:

This is computed here as the (reversed) characteristic polynomial of a square matrix of size twice the number of edges, related to the adjacency matrix of the line graph, see for example Proposition 9 in [SS2008] and Def. 4.1 in [Ter2011].

The graph is first replaced by its 2-core, as this does not change the Ihara zeta function.

EXAMPLES:

```
sage: G = graphs.CompleteGraph(4)
sage: factor(G.ihara_zeta_function_inverse())
(2*t - 1) * (t + 1)^2 * (t - 1)^3 * (2*t^2 + t + 1)^3

sage: G = graphs.CompleteGraph(5)
sage: factor(G.ihara_zeta_function_inverse())
(-1) * (3*t - 1) * (t + 1)^5 * (t - 1)^6 * (3*t^2 + t + 1)^4

sage: G = graphs.PetersenGraph()
sage: factor(G.ihara_zeta_function_inverse())
(-1) * (2*t - 1) * (t + 1)^5 * (t - 1)^6 * (2*t^2 + 2*t + 1)^4
* (2*t^2 - t + 1)^5

sage: G = graphs.RandomTree(10)
sage: G.ihara_zeta_function_inverse()
1
```

REFERENCES:

[HST2001]

independent_set(*algorithm='Cliquer'*, *value_only=False*, *reduction_rules=True*, *solver=None*, *verbosity=0*)

Return a maximum independent set.

An independent set of a graph is a set of pairwise non-adjacent vertices. A maximum independent set is an independent set of maximum cardinality. It induces an empty subgraph.

Equivalently, an independent set is defined as the complement of a vertex cover.

For more information, see the Wikipedia article Independent_set_(graph_theory) and the Wikipedia article Vertex_cover.

INPUT:

- `algorithm` – the algorithm to be used
  - If `algorithm = "Cliquer"` (default), the problem is solved using Cliquer [NO2003].

    (see the *Cliquer modules*)
  - If `algorithm = "MILP"`, the problem is solved through a Mixed Integer Linear Program.

    (see `MixedIntegerLinearProgram`)
- If `algorithm = "mcqd"`, uses the MCQD solver (http://www.sicmm.org/~konc/maxclique/). Note that the MCQD package must be installed.
- `value_only` – boolean (default: `False`); if set to `True`, only the size of a maximum independent set is returned. Otherwise, a maximum independent set is returned as a list of vertices.
- `reduction_rules` – (default: `True`); specify if the reductions rules from kernelization must be applied as pre-processing or not. See [ACFLSS04] for more details. Note that depending on the instance, it might be faster to disable reduction rules.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbosity` – non-negative integer (default: `0`); set the level of verbosity you want from the linear program solver. Since the problem of computing an independent set is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver. This option is only useful if `algorithm="MILP"`.

Note: While Cliquer/MCAD are usually (and by far) the most efficient implementations, the MILP formulation sometimes proves faster on very "symmetrical" graphs.

EXAMPLES:

Using Cliquer:

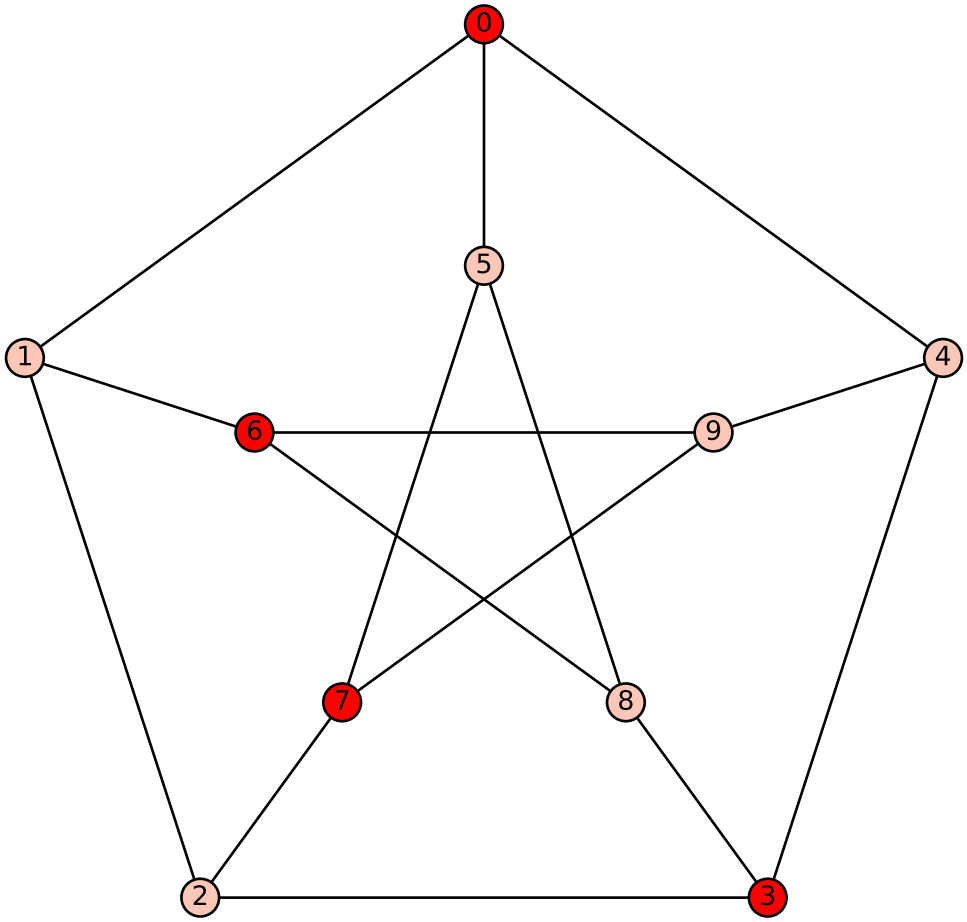
```
sage: C = graphs.PetersenGraph()
sage: C.independent_set()
[0, 3, 6, 7]
```

As a linear program:

```
sage: C = graphs.PetersenGraph()
sage: len(C.independent_set(algorithm="MILP"))
4
```

independent_set_of_representatives (*family*, *solver=None*, *verbose=0*)

Return an independent set of representatives.


0
1
2
3
4
5
6
7
8
9

Given a graph G and a family $F = \{F_i : i \in [1, ..., k]\}$ of subsets of `g.vertices()`, an Independent Set of Representatives (ISR) is an assignation of a vertex $v_i \in F_i$ to each set F_i such that $v_i ! = v_j$ if $i < j$ (they are representatives) and the set $\cup_i v_i$ is an independent set in G.

It generalizes, for example, graph coloring and graph list coloring.

(See [ABZ2007] for more information.)

INPUT:

- `family` – A list of lists defining the family F (actually, a Family of subsets of `G.vertices()`).
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

- A list whose i^{th} element is the representative of the i^{th} element of the `family` list. If there is no ISR, `None` is returned.

EXAMPLES:

For a bipartite graph missing one edge, the solution is as expected:

```
sage: g = graphs.CompleteBipartiteGraph(3,3)
sage: g.delete_edge(1,4)
sage: g.independent_set_of_representatives([[0,1,2],[3,4,5]])
[1, 4]
```

The Petersen Graph is 3-colorable, which can be expressed as an independent set of representatives problem : take 3 disjoint copies of the Petersen Graph, each one representing one color. Then take as a partition of the set of vertices the family defined by the three copies of each vertex. The ISR of such a family defines a 3-coloring:

```
sage: g = 3 * graphs.PetersenGraph()
sage: n = g.order()/3
sage: f = [[i,i+n,i+2*n] for i in range(n)]
sage: isr = g.independent_set_of_representatives(f)
sage: c = [floor(i/n) for i in isr]
sage: color_classes = [[],[],[]]
sage: for v,i in enumerate(c):
....:   color_classes[i].append(v)
sage: for classs in color_classes:
....:   g.subgraph(classs).size() == 0
True
True
True
```

is_apex()

Test if the graph is apex.

A graph is apex if it can be made planar by the removal of a single vertex. The deleted vertex is called `an apex` of the graph, and a graph may have more than one apex. For instance, in the minimal nonplanar graphs K_5 or $K_{3,3}$, every vertex is an apex. The apex graphs include graphs that are themselves planar, in which case again every vertex is an apex. The null graph is also counted as an apex graph even though it has no vertex to remove. If the graph is not connected, we say that it is apex if it has at most one non planar connected component and that this component is apex. See the Wikipedia article Apex_graph for more information.

See also:

- *apex_vertices()*
- *is_planar()*

EXAMPLES:

K_5 and $K_{3,3}$ are apex graphs, and each of their vertices is an apex:

```
sage: G = graphs.CompleteGraph(5)
sage: G.is_apex()
True
sage: G = graphs.CompleteBipartiteGraph(3,3)
sage: G.is_apex()
True
```

The Petersen graph is not apex:

```
sage: G = graphs.PetersenGraph()
sage: G.is_apex()
False
```

A graph is apex if all its connected components are apex, but at most one is not planar:

```
sage: M = graphs.Grid2dGraph(3,3)
sage: K5 = graphs.CompleteGraph(5)
sage: (M+K5).is_apex()
True
sage: (M+K5+K5).is_apex()
False
```

is_arc_transitive()

Check if self is an arc-transitive graph

A graph is arc-transitive if its automorphism group acts transitively on its pairs of adjacent vertices.

Equivalently, if there exists for any pair of edges $uv, u'v' \in E(G)$ an automorphism ϕ_1 of G such that $\phi_1(u) = u'$ and $\phi_1(v) = v'$, as well as another automorphism ϕ_2 of G such that $\phi_2(u) = v'$ and $\phi_2(v) = u'$

See also:

- Wikipedia article arc-transitive_graph
- *is_edge_transitive()*
- *is_half_transitive()*
- *is_semi_symmetric()*

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.is_arc_transitive()
True
sage: G = graphs.GrayGraph()
sage: G.is_arc_transitive()
False
```

is_asteroidal_triple_free(*G*, *certificate=False*)

Test if the input graph is asteroidal triple-free

An independent set of three vertices such that each pair is joined by a path that avoids the neighborhood of the third one is called an *asteroidal triple*. A graph is asteroidal triple-free (AT-free) if it contains no asteroidal triples. See the *module's documentation* for more details.

This method returns `True` is the graph is AT-free and `False` otherwise.

INPUT:

- `G` – a Graph
- `certificate` – boolean (default: `False`); by default, this method returns `True` if the graph is asteroidal triple-free and `False` otherwise. When `certificate==True`, this method returns in addition a list of three vertices forming an asteroidal triple if such a triple is found, and the empty list otherwise.

EXAMPLES:

The complete graph is AT-free, as well as its line graph:

```
sage: G = graphs.CompleteGraph(5)
sage: G.is_asteroidal_triple_free()
True
sage: G.is_asteroidal_triple_free(certificate=True)
(True, [])
sage: LG = G.line_graph()
sage: LG.is_asteroidal_triple_free()
True
sage: LLG = LG.line_graph()
sage: LLG.is_asteroidal_triple_free()
False
```

The PetersenGraph is not AT-free:

```
sage: from sage.graphs.asteroidal_triples import *
sage: G = graphs.PetersenGraph()
sage: G.is_asteroidal_triple_free()
False
sage: G.is_asteroidal_triple_free(certificate=True)
(False, [0, 2, 6])
```

is_biconnected()

Test if the graph is biconnected.

A biconnected graph is a connected graph on two or more vertices that is not broken into disconnected pieces by deleting any single vertex.

See also:

- *is_connected()*
- *blocks_and_cut_vertices()*
- *blocks_and_cuts_tree()*
- Wikipedia article Biconnected_graph

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.is_biconnected()
True
sage: G.add_path([0,'a','b'])
sage: G.is_biconnected()
False
sage: G.add_edge('b', 1)
sage: G.is_biconnected()
True
```

is_block_graph()

Return whether this graph is a block graph.

A block graph is a connected graph in which every biconnected component (block) is a clique.

See also:

- Wikipedia article Block_graph for more details on these graphs
- *RandomBlockGraph()* – generator of random block graphs
- *blocks_and_cut_vertices()*
- *blocks_and_cuts_tree()*

EXAMPLES:

```
sage: G = graphs.RandomBlockGraph(6, 2, kmax=4)
sage: G.is_block_graph()
True
sage: from sage.graphs.isgci import graph_classes
sage: G in graph_classes.Block
True
sage: graphs.CompleteGraph(4).is_block_graph()
True
sage: graphs.RandomTree(6).is_block_graph()
True
sage: graphs.PetersenGraph().is_block_graph()
False
sage: Graph(4).is_block_graph()
False
```

is_cactus()

Check whether the graph is cactus graph.

A graph is called *cactus graph* if it is connected and every pair of simple cycles have at most one common vertex.

There are other definitions, see the Wikipedia article Cactus_graph.

EXAMPLES:

```
sage: g = Graph({1: [2], 2: [3, 4], 3: [4, 5, 6, 7], 8: [3, 5], 9: [6, 7]})
sage: g.is_cactus()
True

sage: c6 = graphs.CycleGraph(6)
sage: naphthalene = c6 + c6
sage: naphthalene.is_cactus()  # Not connected
```

(continues on next page)

(continued from previous page)

```
False
sage: naphthalene.merge_vertices([0, 6])
sage: naphthalene.is_cactus()
True
sage: naphthalene.merge_vertices([1, 7])
sage: naphthalene.is_cactus()
False
```

is_cartesian_product(*g*, *certificate=False*, *relabeling=False*)

Test whether the graph is a Cartesian product.

INPUT:

- `certificate` – boolean (default: `False`); if `certificate = False` (default) the method only returns `True` or `False` answers. If `certificate = True`, the `True` answers are replaced by the list of the factors of the graph.
- `relabeling` – boolean (default: `False`); if `relabeling = True` (implies `certificate = True`), the method also returns a dictionary associating to each vertex its natural coordinates as a vertex of a product graph. If *g* is not a Cartesian product, `None` is returned instead.

See also:

- *sage.graphs.generic_graph.GenericGraph.cartesian_product()*
- *graph_products* – a module on graph products.

Note: This algorithm may run faster whenever the graph's vertices are integers (see *relabel()*). Give it a try if it is too slow !

EXAMPLES:

The Petersen graph is prime:

```
sage: from sage.graphs.graph_decompositions.graph_products import is_
↪cartesian_product
sage: g = graphs.PetersenGraph()
sage: is_cartesian_product(g)
False
```

A 2d grid is the product of paths:

```
sage: g = graphs.Grid2dGraph(5,5)
sage: p1, p2 = is_cartesian_product(g, certificate = True)
sage: p1.is_isomorphic(graphs.PathGraph(5))
True
sage: p2.is_isomorphic(graphs.PathGraph(5))
True
```

Forgetting the graph's labels, then finding them back:

```
sage: g.relabel()
sage: b,D = g.is_cartesian_product(g, relabeling=True)
sage: b
True
sage: D  # random isomorphism
```

(continues on next page)

(continued from previous page)

```
{0: (20, 0), 1: (20, 1), 2: (20, 2), 3: (20, 3), 4: (20, 4),
 5: (15, 0), 6: (15, 1), 7: (15, 2), 8: (15, 3), 9: (15, 4),
 10: (10, 0), 11: (10, 1), 12: (10, 2), 13: (10, 3), 14: (10, 4),
 15: (5, 0), 16: (5, 1), 17: (5, 2), 18: (5, 3), 19: (5, 4),
 20: (0, 0), 21: (0, 1), 22: (0, 2), 23: (0, 3), 24: (0, 4)}
```

And of course, we find the factors back when we build a graph from a product:

```
sage: g = graphs.PetersenGraph().cartesian_product(graphs.CycleGraph(3))
sage: g1, g2 = is_cartesian_product(g, certificate = True)
sage: any( x.is_isomorphic(graphs.PetersenGraph()) for x in [g1,g2])
True
sage: any( x.is_isomorphic(graphs.CycleGraph(3)) for x in [g1,g2])
True
```

is_circumscribable(*solver='ppl'*, *verbose=0*)

Test whether the graph is the graph of a circumscribed polyhedron.

A polyhedron is circumscribed if all of its facets are tangent to a sphere. By a theorem of Rivin ([HRS1993]), this can be checked by solving a linear program that assigns weights between 0 and 1/2 on each edge of the polyhedron, so that the weights on any face add to exactly one and the weights on any non-facial cycle add to more than one. If and only if this can be done, the polyhedron can be circumscribed.

INPUT:

- `solver` – (default: `"ppl"`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: C = graphs.CubeGraph(3)
sage: C.is_circumscribable()
True

sage: O = graphs.OctahedralGraph()
sage: O.is_circumscribable()
True

sage: TT = polytopes.truncated_tetrahedron().graph()
sage: TT.is_circumscribable()
False
```

Stellating in a face of the octahedral graph is not circumscribable:

```
sage: f = set(flatten(choice(O.faces())))
sage: O.add_edges([[6, i] for i in f])
sage: O.is_circumscribable()
False
```

See also:

- *is_polyhedral()*
- *is_inscribable()*

Todo: Allow the use of other, inexact but faster solvers.

is_cograph()
Check whether the graph is cograph.

A cograph is defined recursively: the single-vertex graph is cograph, complement of cograph is cograph, and disjoint union of two cographs is cograph. There are many other characterizations, see the Wikipedia article Cograph.

EXAMPLES:

```
sage: graphs.HouseXGraph().is_cograph()
True
sage: graphs.HouseGraph().is_cograph()
False
```

Todo: Implement faster recognition algorithm, as for instance the linear time recognition algorithm using LexBFS proposed in [Bre2008].

is_comparability(*g*, *algorithm='greedy'*, *certificate=False*, *check=True*, *solver=None*, *verbose=0*)
Tests whether the graph is a comparability graph

INPUT:

- `algorithm` – choose the implementation used to do the test.
  - `"greedy"` – a greedy algorithm (see the documentation of the *comparability module*).
  - `"MILP"` – a Mixed Integer Linear Program formulation of the problem. Beware, for this implementation is unable to return negative certificates ! When `certificate = True`, negative certificates are always equal to `None`. True certificates are valid, though.
- `certificate` (boolean) – whether to return a certificate. *Yes*-answers the certificate is a transitive orientation of G, and a *no* certificates is an odd cycle of sequentially forcing edges.
- `check` (boolean) – whether to check that the yes-certificates are indeed transitive. As it is very quick compared to the rest of the operation, it is enabled by default.
- `solver` – (default: `None`); Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: from sage.graphs.comparability import is_comparability
sage: g = graphs.PetersenGraph()
sage: is_comparability(g)
False
sage: is_comparability(graphs.CompleteGraph(5), certificate=True)
(True, Digraph on 5 vertices)
```

is_directed()
Since graph is undirected, returns False.

EXAMPLES:

```
sage: Graph().is_directed()
False
```

is_distance_regular(*G*, *parameters=False*)

Test if the graph is distance-regular

A graph G is distance-regular if for any integers j, k the value of $|\{x : d_G(x,u) = j, x \in V(G)\} \cap \{y : d_G(y,v) = j, y \in V(G)\}|$ is constant for any two vertices $u, v \in V(G)$ at distance i from each other. In particular G is regular, of degree b_0 (see below), as one can take $u = v$.

Equivalently a graph is distance-regular if there exist integers b_i, c_i such that for any two vertices u, v at distance i we have

- $b_i = |\{x : d_G(x,u) = i+1, x \in V(G)\} \cap N_G(v)\}|$, $0 \leq i \leq d-1$
- $c_i = |\{x : d_G(x,u) = i-1, x \in V(G)\} \cap N_G(v)\}|$, $1 \leq i \leq d$,

where d is the diameter of the graph. For more information on distance-regular graphs, see the Wikipedia article Distance-regular_graph.

INPUT:

- `parameters` – boolean (default: `False`); if set to `True`, the function returns the pair `(b, c)` of lists of integers instead of a boolean answer (see the definition above)

See also:

- *is_regular()*
- *is_strongly_regular()*

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.is_distance_regular()
True
sage: g.is_distance_regular(parameters = True)
([3, 2, None], [None, 1, 1])
```

Cube graphs, which are not strongly regular, are a bit more interesting:

```
sage: graphs.CubeGraph(4).is_distance_regular()
True
sage: graphs.OddGraph(5).is_distance_regular()
True
```

Disconnected graph:

```
sage: (2*graphs.CubeGraph(4)).is_distance_regular()
True
```

is_dominating(*G*, *dom*, *focus=None*)

Check whether `dom` is a dominating set of `G`.

We say that a set D of vertices of a graph G dominates a set S if every vertex of S either belongs to D or is adjacent to a vertex of D. Also, D is a dominating set of G if it dominates $V(G)$.

INPUT:

- `dom` – iterable of vertices of `G`; the vertices of the supposed dominating set.

- `focus` – iterable of vertices of `G` (default: `None`); if specified, this method checks instead if `dom` dominates the vertices in `focus`.

EXAMPLES:

```
sage: g = graphs.CycleGraph(5)
sage: g.is_dominating([0,1], [4, 2])
True

sage: g.is_dominating([0,1])
False
```

is_edge_transitive()

Check if self is an edge transitive graph.

A graph is edge-transitive if its automorphism group acts transitively on its edge set.

Equivalently, if there exists for any pair of edges $uv, u'v' \in E(G)$ an automorphism ϕ of G such that $\phi(uv) = u'v'$ (note this does not necessarily mean that $\phi(u) = u'$ and $\phi(v) = v'$).

See also:

- Wikipedia article Edge-transitive_graph
- *is_arc_transitive()*
- *is_half_transitive()*
- *is_semi_symmetric()*

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.is_edge_transitive()
True
sage: C = graphs.CubeGraph(3)
sage: C.is_edge_transitive()
True
sage: G = graphs.GrayGraph()
sage: G.is_edge_transitive()
True
sage: P = graphs.PathGraph(4)
sage: P.is_edge_transitive()
False
```

is_even_hole_free(*certificate=False*)

Tests whether `self` contains an induced even hole.

A Hole is a cycle of length at least 4 (included). It is said to be even (resp. odd) if its length is even (resp. odd).

Even-hole-free graphs always contain a bisimplicial vertex, which ensures that their chromatic number is at most twice their clique number [ACHRS2008].

INPUT:

- `certificate` – boolean (default: `False`); when `certificate = False`, this method only returns `True` or `False`. If `certificate = True`, the subgraph found is returned instead of `False`.

EXAMPLES:

Is the Petersen Graph even-hole-free

```
sage: g = graphs.PetersenGraph()
sage: g.is_even_hole_free()
False
```

As any chordal graph is hole-free, interval graphs behave the same way:

```
sage: g = graphs.RandomIntervalGraph(20)
sage: g.is_even_hole_free()
True
```

It is clear, though, that a random Bipartite Graph which is not a forest has an even hole:

```
sage: g = graphs.RandomBipartite(10, 10, .5)
sage: g.is_even_hole_free() and not g.is_forest()
False
```

We can check the certificate returned is indeed an even cycle:

```
sage: if not g.is_forest():
....:     cycle = g.is_even_hole_free(certificate=True)
....:     if cycle.order() % 2 == 1:
....:         print("Error !")
....:     if not cycle.is_isomorphic(
....:             graphs.CycleGraph(cycle.order())):
....:         print("Error !")
...
sage: print("Everything is Fine !")
Everything is Fine !
```

is_forest(*certificate=False*, *output='vertex'*)

Tests if the graph is a forest, i.e. a disjoint union of trees.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate. The method only returns boolean answers when `certificate = False` (default). When it is set to `True`, it either answers `(True, None)` when the graph is a forest or `(False, cycle)` when it contains a cycle.
- `output` – either `'vertex'` (default) or `'edge'`; whether the certificate is given as a list of vertices (`output = 'vertex'`) or a list of edges (`output = 'edge'`).

EXAMPLES:

```
sage: seven_acre_wood = sum(graphs.trees(7), Graph())
sage: seven_acre_wood.is_forest()
True
```

With certificates:

```
sage: g = graphs.RandomTree(30)
sage: g.is_forest(certificate=True)
(True, None)
sage: (2*g + graphs.PetersenGraph() + g).is_forest(certificate=True)
(False, [62, 63, 68, 66, 61])
```

is_half_transitive()

Check if self is a half-transitive graph.

A graph is half-transitive if it is both vertex and edge transitive but not arc-transitive.

See also:

- Wikipedia article half-transitive_graph
- *is_edge_transitive()*
- *is_arc_transitive()*
- *is_semi_symmetric()*

EXAMPLES:

The Petersen Graph is not half-transitive:

```
sage: P = graphs.PetersenGraph()
sage: P.is_half_transitive()
False
```

The smallest half-transitive graph is the Holt Graph:

```
sage: H = graphs.HoltGraph()
sage: H.is_half_transitive()
True
```

is_inscribable(*solver='ppl'*, *verbose=0*)

Test whether the graph is the graph of an inscribed polyhedron.

A polyhedron is inscribed if all of its vertices are on a sphere. This is dual to the notion of circumscribed polyhedron: A Polyhedron is inscribed if and only if its polar dual is circumscribed and hence a graph is inscribable if and only if its planar dual is circumscribable.

INPUT:

- `solver` – (default: `"ppl"`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: H = graphs.HerschelGraph()
sage: H.is_inscribable()                # long time (> 1 sec)
False
sage: H.planar_dual().is_inscribable() # long time (> 1 sec)
True

sage: C = graphs.CubeGraph(3)
sage: C.is_inscribable()
True
```

Cutting off a vertex from the cube yields an uninscribable graph:

```
sage: C = graphs.CubeGraph(3)
sage: v = next(C.vertex_iterator())
sage: triangle = [_ + v for _ in C.neighbors(v)]
sage: C.add_edges(Combinations(triangle, 2))
sage: C.add_edges(zip(triangle, C.neighbors(v)))
sage: C.delete_vertex(v)
```

(continues on next page)

(continued from previous page)

```
sage: C.is_inscribable()
False
```

Breaking a face of the cube yields an uninscribable graph:

```
sage: C = graphs.CubeGraph(3)
sage: face = choice(C.faces())
sage: C.add_edge([face[0][0], face[2][0]])
sage: C.is_inscribable()
False
```

See also:

- *is_polyhedral()*
- *is_circumscribable()*

is_line_graph(*g*, *certificate=False*)

Tests wether the graph is a line graph.

INPUT:

- `certificate` (boolean) – whether to return a certificate along with the boolean result. Here is what happens when `certificate = True`:
  - If the graph is not a line graph, the method returns a pair `(b, subgraph)` where `b` is `False` and `subgraph` is a subgraph isomorphic to one of the 9 forbidden induced subgraphs of a line graph.
  - If the graph is a line graph, the method returns a triple `(b,R,isom)` where `b` is `True`, `R` is a graph whose line graph is the graph given as input, and `isom` is a map associating an edge of `R` to each vertex of the graph.

Note: This method wastes a bit of time when the input graph is not connected. If you have performance in mind, it is probably better to only feed it with connected graphs only.

See also:

- The *line_graph* module.
- *line_graph_forbidden_subgraphs()* – the forbidden subgraphs of a line graph.
- *line_graph()*

EXAMPLES:

A complete graph is always the line graph of a star:

```
sage: graphs.CompleteGraph(5).is_line_graph()
True
```

The Petersen Graph not being claw-free, it is not a line graph:

```
sage: graphs.PetersenGraph().is_line_graph()
False
```

This is indeed the subgraph returned:

```
sage: C = graphs.PetersenGraph().is_line_graph(certificate=True)[1]
sage: C.is_isomorphic(graphs.ClawGraph())
True
```

The house graph is a line graph:

```
sage: g = graphs.HouseGraph()
sage: g.is_line_graph()
True
```

But what is the graph whose line graph is the house ?:

```
sage: is_line, R, isom = g.is_line_graph(certificate=True)
sage: R.sparse6_string()
':DaHI~'
sage: R.show()
sage: isom
{0: (0, 1), 1: (0, 2), 2: (1, 3), 3: (2, 3), 4: (3, 4)}
```

is_long_antihole_free(*g*, *certificate=False*)

Tests whether the given graph contains an induced subgraph that is isomorphic to the complement of a cycle of length at least 5.

INPUT:

- `certificate` – boolean (default: `False`)

  Whether to return a certificate. When `certificate = True`, then the function returns

  - `(False, Antihole)` if `g` contains an induced complement of a cycle of length at least 5 returned as `Antihole`.
  - `(True, [])` if `g` does not contain an induced complement of a cycle of length at least 5. For this case it is not known how to provide a certificate.

  When `certificate = False`, the function returns just `True` or `False` accordingly.

ALGORITHM:

This algorithm tries to find a cycle in the graph of all induced $\overline{P_4}$ of g, where two copies $\overline{P}$ and $\overline{P'}$ of $\overline{P_4}$ are adjacent if there exists a (not necessarily induced) copy of $\overline{P_5} = u_1u_2u_3u_4u_5$ such that $\overline{P} = u_1u_2u_3u_4$ and $\overline{P'} = u_2u_3u_4u_5$.

This is done through a depth-first-search. For efficiency, the auxiliary graph is constructed on-the-fly and never stored in memory.

The run time of this algorithm is $O(m^2)$ [NP2007] (where m is the number of edges of the graph).

EXAMPLES:

The Petersen Graph contains an antihole:

```
sage: g = graphs.PetersenGraph()
sage: g.is_long_antihole_free()
False
```

The complement of a cycle is an antihole:

```
sage: g = graphs.CycleGraph(6).complement()
sage: r,a = g.is_long_antihole_free(certificate=True)
sage: r
```

(continues on next page)

(continued from previous page)

```
False
sage: a.complement().is_isomorphic(graphs.CycleGraph(6))
True
```

is_long_hole_free(*g*, *certificate=False*)

Tests whether `g` contains an induced cycle of length at least 5.

INPUT:

- `certificate` – boolean (default: `False`)

  Whether to return a certificate. When `certificate = True`, then the function returns

  - `(True, [])` if `g` does not contain such a cycle. For this case, it is not known how to provide a certificate.
  - `(False, Hole)` if `g` contains an induced cycle of length at least 5. `Hole` returns this cycle.

  If `certificate = False`, the function returns just `True` or `False` accordingly.

ALGORITHM:

This algorithm tries to find a cycle in the graph of all induced P_4 of g, where two copies P and P' of P_4 are adjacent if there exists a (not necessarily induced) copy of $P_5 = u_1u_2u_3u_4u_5$ such that $P = u_1u_2u_3u_4$ and $P' = u_2u_3u_4u_5$.

This is done through a depth-first-search. For efficiency, the auxiliary graph is constructed on-the-fly and never stored in memory.

The run time of this algorithm is $O(m^2)$ [NP2007] (where m is the number of edges of the graph) .

EXAMPLES:

The Petersen Graph contains a hole:

```
sage: g = graphs.PetersenGraph()
sage: g.is_long_hole_free()
False
```

The following graph contains a hole, which we want to display:

```
sage: g = graphs.FlowerSnark()
sage: r,h = g.is_long_hole_free(certificate=True)
sage: r
False
sage: Graph(h).is_isomorphic(graphs.CycleGraph(h.order()))
True
```

is_odd_hole_free(*certificate=False*)

Tests whether `self` contains an induced odd hole.

A Hole is a cycle of length at least 4 (included). It is said to be even (resp. odd) if its length is even (resp. odd).

It is interesting to notice that while it is polynomial to check whether a graph has an odd hole or an odd antihole [CCLSV2005], it is not known whether testing for one of these two cases independently is polynomial too.

INPUT:

- `certificate` – boolean (default: `False`); when `certificate = False`, this method only returns `True` or `False`. If `certificate = True`, the subgraph found is returned instead of `False`.

EXAMPLES:

Is the Petersen Graph odd-hole-free

```
sage: g = graphs.PetersenGraph()
sage: g.is_odd_hole_free()
False
```

Which was to be expected, as its girth is 5

```
sage: g.girth()
5
```

We can check the certificate returned is indeed a 5-cycle:

```
sage: cycle = g.is_odd_hole_free(certificate=True)
sage: cycle.is_isomorphic(graphs.CycleGraph(5))
True
```

As any chordal graph is hole-free, no interval graph has an odd hole:

```
sage: g = graphs.RandomIntervalGraph(20)
sage: g.is_odd_hole_free()
True
```

is_overfull()

Tests whether the current graph is overfull.

A graph G on n vertices and m edges is said to be overfull if:

- n is odd
- It satisfies $2m > (n-1)\Delta(G)$, where $\Delta(G)$ denotes the maximum degree among all vertices in G.

An overfull graph must have a chromatic index of $\Delta(G)+1$.

EXAMPLES:

A complete graph of order $n > 1$ is overfull if and only if n is odd:

```
sage: graphs.CompleteGraph(6).is_overfull()
False
sage: graphs.CompleteGraph(7).is_overfull()
True
sage: graphs.CompleteGraph(1).is_overfull()
False
```

The claw graph is not overfull:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: g = graphs.ClawGraph()
sage: g
Claw graph: Graph on 4 vertices
sage: edge_coloring(g, value_only=True)
3
sage: g.is_overfull()
False
```

The Holt graph is an example of a overfull graph:

```
sage: G = graphs.HoltGraph()
sage: G.is_overfull()
True
```

Checking that all complete graphs K_n for even $0 \leq n \leq 100$ are not overfull:

```
sage: def check_overfull_Kn_even(n):
....:     i = 0
....:     while i <= n:
....:         if graphs.CompleteGraph(i).is_overfull():
....:             print("A complete graph of even order cannot be overfull.")
....:             return
....:         i += 2
....:     print("Complete graphs of even order up to %s are not overfull." %␣
↪n)
...
sage: check_overfull_Kn_even(100)  # long time
Complete graphs of even order up to 100 are not overfull.
```

The null graph, i.e. the graph with no vertices, is not overfull:

```
sage: Graph().is_overfull()
False
sage: graphs.CompleteGraph(0).is_overfull()
False
```

Checking that all complete graphs K_n for odd $1 < n \leq 100$ are overfull:

```
sage: def check_overfull_Kn_odd(n):
....:     i = 3
....:     while i <= n:
....:         if not graphs.CompleteGraph(i).is_overfull():
....:             print("A complete graph of odd order > 1 must be overfull.")
....:             return
....:         i += 2
....:     print("Complete graphs of odd order > 1 up to %s are overfull." % n)
...
sage: check_overfull_Kn_odd(100)  # long time
Complete graphs of odd order > 1 up to 100 are overfull.
```

The Petersen Graph, though, is not overfull while its chromatic index is $\Delta + 1$:

```
sage: g = graphs.PetersenGraph()
sage: g.is_overfull()
False
sage: from sage.graphs.graph_coloring import edge_coloring
sage: max(g.degree()) + 1 ==  edge_coloring(g, value_only=True)
True
```

is_partial_cube(*G*, *certificate=False*)

Test whether the given graph is a partial cube.

A partial cube is a graph that can be isometrically embedded into a hypercube, i.e., its vertices can be labelled with (0,1)-vectors of some fixed length such that the distance between any two vertices in the graph equals the Hamming distance of their labels.

Originally written by D. Eppstein for the PADS library (http://www.ics.uci.edu/~eppstein/PADS/), see also

[Epp2008]. The algorithm runs in $O(n^2)$ time, where n is the number of vertices. See the documentation of *partial_cube* for an overview of the algorithm.

INPUT:

- `certificate` – boolean (default: `False`); this function returns `True` or `False` according to the graph, when `certificate = False`. When `certificate = True` and the graph is a partial cube, the function returns `(True, mapping)`, where `mapping` is an isometric mapping of the vertices of the graph to the vertices of a hypercube ((0, 1)-strings of a fixed length). When `certificate = True` and the graph is not a partial cube, `(False, None)` is returned.

EXAMPLES:

The Petersen graph is not a partial cube:

```
sage: g = graphs.PetersenGraph()
sage: g.is_partial_cube()
False
```

All prisms are partial cubes:

```
sage: g = graphs.CycleGraph(10).cartesian_product(graphs.CompleteGraph(2))
sage: g.is_partial_cube()
True
```

is_perfect(*certificate=False*)

Tests whether the graph is perfect.

A graph G is said to be perfect if $\chi(H) = \omega(H)$ hold for any induced subgraph $H \subseteq_i G$ (and so for G itself, too), where $\chi(H)$ represents the chromatic number of H, and $\omega(H)$ its clique number. The Strong Perfect Graph Theorem [CRST2006] gives another characterization of perfect graphs:

A graph is perfect if and only if it contains no odd hole (cycle on an odd number k of vertices, $k > 3$) nor any odd antihole (complement of a hole) as an induced subgraph.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate.

OUTPUT:

When `certificate = False`, this function returns a boolean value. When `certificate = True`, it returns a subgraph of `self` isomorphic to an odd hole or an odd antihole if any, and `None` otherwise.

EXAMPLES:

A Bipartite Graph is always perfect

```
sage: g = graphs.RandomBipartite(8,4,.5)
sage: g.is_perfect()
True
```

So is the line graph of a bipartite graph:

```
sage: g = graphs.RandomBipartite(4,3,0.7)
sage: g.line_graph().is_perfect() # long time
True
```

As well as the Cartesian product of two complete graphs:

```
sage: g = graphs.CompleteGraph(3).cartesian_product(graphs.CompleteGraph(3))
sage: g.is_perfect()
True
```

Interval Graphs, which are chordal graphs, too

```
sage: g =  graphs.RandomIntervalGraph(7)
sage: g.is_perfect()
True
```

The PetersenGraph, which is triangle-free and has chromatic number 3 is obviously not perfect:

```
sage: g = graphs.PetersenGraph()
sage: g.is_perfect()
False
```

We can obtain an induced 5-cycle as a certificate:

```
sage: g.is_perfect(certificate=True)
Subgraph of (Petersen graph): Graph on 5 vertices
```

is_permutation(*g*, *algorithm='greedy'*, *certificate=False*, *check=True*, *solver=None*, *verbose=0*)

Tests whether the graph is a permutation graph.

For more information on permutation graphs, refer to the documentation of the *comparability module*.

INPUT:

- `algorithm` – choose the implementation used for the subcalls to *is_comparability()*.
  - `"greedy"` – a greedy algorithm (see the documentation of the *comparability module*).
  - `"MILP"` – a Mixed Integer Linear Program formulation of the problem. Beware, for this implementation is unable to return negative certificates ! When `certificate = True`, negative certificates are always equal to `None`. True certificates are valid, though.
- `certificate` (boolean) – whether to return a certificate for the answer given. For `True` answers the certificate is a permutation, for `False` answers it is a no-certificate for the test of comparability or co-comparability.
- `check` (boolean) – whether to check that the permutations returned indeed create the expected Permutation graph. Pretty cheap compared to the rest, hence a good investment. It is enabled by default.
- `solver` – (default: `None`); Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: As the `True` certificate is a `Permutation` object, the segment intersection model of the permutation graph can be visualized through a call to `Permutation.show`.

EXAMPLES:

A permutation realizing the bull graph:

```
sage: from sage.graphs.comparability import is_permutation
sage: g = graphs.BullGraph()
sage: _ , certif = is_permutation(g, certificate=True)
sage: h = graphs.PermutationGraph(*certif)
sage: h.is_isomorphic(g)
True
```

Plotting the realization as an intersection graph of segments:

```
sage: true, perm = is_permutation(g, certificate=True)
sage: p1 = Permutation([nn+1 for nn in perm[0]])
sage: p2 = Permutation([nn+1 for nn in perm[1]])
sage: p = p2 * p1.inverse()
sage: p.show(representation = "braid")
```

is_polyhedral()

Check whether the graph is the graph of the polyhedron.

By a theorem of Steinitz (Satz 43, p. 77 of [St1922]), graphs of three-dimensional polyhedra are exactly the simple 3-vertex-connected planar graphs.

EXAMPLES:

```
sage: C = graphs.CubeGraph(3)
sage: C.is_polyhedral()
True
sage: K33=graphs.CompleteBipartiteGraph(3, 3)
sage: K33.is_polyhedral()
False
sage: graphs.CycleGraph(17).is_polyhedral()
False
sage: [i for i in range(9) if graphs.CompleteGraph(i).is_polyhedral()]
[4]
```

See also:

- *vertex_connectivity()*
- *is_planar()*
- *is_circumscribable()*
- *is_inscribable()*
- Wikipedia article Polyhedral_graph

is_prime(*algorithm='habib'*)

Test whether the current graph is prime.

INPUT:

- `algorithm` – (default: `'tedder'`) specifies the algorithm to use among:
  - `'tedder'` – Use the linear algorithm of [TCHP2008].
  - `'habib'` – Use the $O(n^3)$ algorithm of [HM1979]. This is probably slower, but is much simpler and so possibly less error prone.

A graph is prime if all its modules are trivial (i.e. empty, all of the graph or singletons) – see *modular_decomposition()*.

EXAMPLES:

The Petersen Graph and the Bull Graph are both prime:

```
sage: graphs.PetersenGraph().is_prime()
True
sage: graphs.BullGraph().is_prime()
True
```

Though quite obviously, the disjoint union of them is not:

```
sage: (graphs.PetersenGraph() + graphs.BullGraph()).is_prime()
False
```

is_redundant(*G*, *dom*, *focus=None*)

Check whether `dom` has redundant vertices.

For a graph G and sets D and S of vertices, we say that a vertex $v \in D$ is *redundant* in S if v has no private neighbor with repect to D in S. In other words, there is no vertex in S that is dominated by v but not by $D \setminus \{v\}$.

INPUT:

- `dom` – iterable of vertices of `G`; where we look for redundant vertices.
- `focus` – iterable of vertices of `G` (default: `None`); if specified, this method checks instead whether `dom` has a redundant vertex in `focus`.

Warning: The assumption is made that `focus` (if provided) does not contain repeated vertices.

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: G.is_redundant(['000', '101'], ['011'])
True
sage: G.is_redundant(['000', '101'])
False
```

is_semi_symmetric()

Check if self is semi-symmetric.

A graph is semi-symmetric if it is regular, edge-transitive but not vertex-transitive.

See also:

- Wikipedia article Semi-symmetric_graph
- *is_edge_transitive()*
- *is_arc_transitive()*
- *is_half_transitive()*

EXAMPLES:

The Petersen graph is not semi-symmetric:

```
sage: P = graphs.PetersenGraph()
sage: P.is_semi_symmetric()
False
```

The Gray graph is the smallest possible cubic semi-symmetric graph:

```
sage: G = graphs.GrayGraph()
sage: G.is_semi_symmetric()
True
```

Another well known semi-symmetric graph is the Ljubljana graph:

```
sage: L = graphs.LjubljanaGraph()
sage: L.is_semi_symmetric()
True
```

is_split()

Returns `True` if the graph is a Split graph, `False` otherwise.

A Graph G is said to be a split graph if its vertices $V(G)$ can be partitioned into two sets K and I such that the vertices of K induce a complete graph, and those of I are an independent set.

There is a simple test to check whether a graph is a split graph (see, for instance, the book "Graph Classes, a survey" [BLS1999] page 203) :

Given the degree sequence $d_1 \geq ... \geq d_n$ of G, a graph is a split graph if and only if :

$$\sum_{i=1}^{\omega} d_i = \omega(\omega - 1) + \sum_{i=\omega+1}^{n} d_i$$

where $\omega = max\{i : d_i \geq i - 1\}$.

EXAMPLES:

Split graphs are, in particular, chordal graphs. Hence, The Petersen graph can not be split:

```
sage: graphs.PetersenGraph().is_split()
False
```

We can easily build some "random" split graph by creating a complete graph, and adding vertices only connected to some random vertices of the clique:

```
sage: g = graphs.CompleteGraph(10)
sage: sets = Subsets(Set(range(10)))
sage: for i in range(10, 25):
....:     g.add_edges([(i,k) for k in sets.random_element()])
sage: g.is_split()
True
```

Another characterisation of split graph states that a graph is a split graph if and only if does not contain the 4-cycle, 5-cycle or $2K_2$ as an induced subgraph. Hence for the above graph we have:

```
sage: forbidden_subgraphs = [graphs.CycleGraph(4), graphs.CycleGraph(5), 2 *␣
↪graphs.CompleteGraph(2)]
sage: sum(g.subgraph_search_count(H,induced=True) for H in forbidden_
↪subgraphs)
0
```

is_strongly_regular(*g*, *parameters=False*)

Check whether the graph is strongly regular.

A simple graph G is said to be strongly regular with parameters (n, k, λ, μ) if and only if:

- G has n vertices

- G is k-regular
- Any two adjacent vertices of G have λ common neighbors
- Any two non-adjacent vertices of G have μ common neighbors

By convention, the complete graphs, the graphs with no edges and the empty graph are not strongly regular.

See the Wikipedia article Strongly regular graph.

INPUT:

- `parameters` – boolean (default: `False`); whether to return the quadruple (n, k, λ, μ). If `parameters = False` (default), this method only returns `True` and `False` answers. If `parameters = True`, the `True` answers are replaced by quadruples (n, k, λ, μ). See definition above.

EXAMPLES:

Petersen's graph is strongly regular:

```
sage: g = graphs.PetersenGraph()
sage: g.is_strongly_regular()
True
sage: g.is_strongly_regular(parameters=True)
(10, 3, 0, 1)
```

And Clebsch's graph is too:

```
sage: g = graphs.ClebschGraph()
sage: g.is_strongly_regular()
True
sage: g.is_strongly_regular(parameters=True)
(16, 5, 0, 2)
```

But Chvatal's graph is not:

```
sage: g = graphs.ChvatalGraph()
sage: g.is_strongly_regular()
False
```

Complete graphs are not strongly regular. (trac ticket #14297)

```
sage: g = graphs.CompleteGraph(5)
sage: g.is_strongly_regular()
False
```

Completements of complete graphs are not strongly regular:

```
sage: g = graphs.CompleteGraph(5).complement()
sage: g.is_strongly_regular()
False
```

The empty graph is not strongly regular:

```
sage: g = graphs.EmptyGraph()
sage: g.is_strongly_regular()
False
```

If the input graph has loops or multiedges an exception is raised:

```
sage: Graph([(1,1),(2,2)],loops=True).is_strongly_regular()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with
loops. Perhaps this method can be updated to handle them, but in the
meantime if you want to use it please disallow loops using
allow_loops().

sage: Graph([(1,2),(1,2)],multiedges=True).is_strongly_regular()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with
multiedges. Perhaps this method can be updated to handle them, but in
the meantime if you want to use it please disallow multiedges using
allow_multiple_edges().
```

is_tree(*certificate=False*, *output='vertex'*)

Tests if the graph is a tree

The empty graph is defined to be not a tree.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate. The method only returns boolean answers when `certificate = False` (default). When it is set to `True`, it either answers `(True, None)` when the graph is a tree or `(False, cycle)` when it contains a cycle. It returns `(False, None)` when the graph is empty or not connected.
- `output` – either `'vertex'` (default) or `'edge'`; whether the certificate is given as a list of vertices (`output = 'vertex'`) or a list of edges (`output = 'edge'`).

When the certificate cycle is given as a list of edges, the edges are given as (v_i, v_{i+1}, l) where $v_1, v_2, \ldots, v_n$ are the vertices of the cycles (in their cyclic order).

EXAMPLES:

```
sage: all(T.is_tree() for T in graphs.trees(15))
True
```

With certificates:

```
sage: g = graphs.RandomTree(30)
sage: g.is_tree(certificate=True)
(True, None)
sage: g.add_edge(10,-1)
sage: g.add_edge(11,-1)
sage: isit, cycle = g.is_tree(certificate=True)
sage: isit
False
sage: -1 in cycle
True
```

One can also ask for the certificate as a list of edges:

```
sage: g = graphs.CycleGraph(4)
sage: g.is_tree(certificate=True, output='edge')
(False, [(3, 2, None), (2, 1, None), (1, 0, None), (0, 3, None)])
```

This is useful for graphs with multiple edges:

```
sage: G = Graph([(1, 2, 'a'), (1, 2, 'b')], multiedges=True)
sage: G.is_tree(certificate=True)
(False, [1, 2])
sage: G.is_tree(certificate=True, output='edge')
(False, [(1, 2, 'a'), (2, 1, 'b')])
```

is_triangle_free(*algorithm='dense_graph'*, *certificate=False*)

Check whether `self` is triangle-free

INPUT:

- `algorithm` – (default: `'dense_graph'`) specifies the algorithm to use among:
  - `'matrix'` – tests if the trace of the adjacency matrix is positive.
  - `'bitset'` – encodes adjacencies into bitsets and uses fast bitset operations to test if the input graph contains a triangle. This method is generally faster than standard matrix multiplication.
  - `'dense_graph'` – use the implementation of *sage.graphs.base.static_dense_graph*
- `certificate` – boolean (default: `False`); whether to return a triangle if one is found. This parameter is ignored when `algorithm` is `'matrix'`.

EXAMPLES:

The Petersen Graph is triangle-free:

```
sage: g = graphs.PetersenGraph()
sage: g.is_triangle_free()
True
```

or a complete Bipartite Graph:

```
sage: G = graphs.CompleteBipartiteGraph(5,6)
sage: G.is_triangle_free(algorithm='matrix')
True
sage: G.is_triangle_free(algorithm='bitset')
True
sage: G.is_triangle_free(algorithm='dense_graph')
True
```

a tripartite graph, though, contains many triangles:

```
sage: G = (3 * graphs.CompleteGraph(5)).complement()
sage: G.is_triangle_free(algorithm='matrix')
False
sage: G.is_triangle_free(algorithm='bitset')
False
sage: G.is_triangle_free(algorithm='dense_graph')
False
```

Asking for a certificate:

```
sage: K4 = graphs.CompleteGraph(4)
sage: K4.is_triangle_free(algorithm='dense_graph', certificate=True)
(False, [0, 1, 2])
sage: K4.is_triangle_free(algorithm='bitset', certificate=True)
(False, [0, 1, 2])
```

is_triconnected(*G*)

Check whether the graph is triconnected.

A triconnected graph is a connected graph on 3 or more vertices that is not broken into disconnected pieces by deleting any pair of vertices.

EXAMPLES:

The Petersen graph is triconnected:

```
sage: G = graphs.PetersenGraph()
sage: G.is_triconnected()
True
```

But a 2D grid is not:

```
sage: G = graphs.Grid2dGraph(3, 3)
sage: G.is_triconnected()
False
```

By convention, a cycle of order 3 is triconnected:

```
sage: G = graphs.CycleGraph(3)
sage: G.is_triconnected()
True
```

But cycles of order 4 and more are not:

```
sage: [graphs.CycleGraph(i).is_triconnected() for i in range(4, 8)]
[False, False, False, False]
```

Comparing different methods on random graphs that are not always triconnected:

```
sage: G = graphs.RandomBarabasiAlbert(50, 3)
sage: G.is_triconnected() == G.vertex_connectivity(k=3)
True
```

See also:

- *is_connected()*
- *is_biconnected()*
- *spqr_tree()*
- Wikipedia article SPQR_tree

is_weakly_chordal(*g*, *certificate=False*)

Tests whether the given graph is weakly chordal, i.e., the graph and its complement have no induced cycle of length at least 5.

INPUT:

- `certificate` – Boolean value (default: `False`) whether to return a certificate. If `certificate = False`, return `True` or `False` according to the graph. If `certificate = True`, return
  - `(False, forbidden_subgraph)` when the graph contains a forbidden subgraph H, this graph is returned.
  - **`(True, [])` when the graph is weakly chordal.** For this case, it is not known how to provide a certificate.

ALGORITHM:

This algorithm checks whether the graph `g` or its complement contain an induced cycle of length at least 5.

Using is_long_hole_free() and is_long_antihole_free() yields a run time of $O(m^2)$ (where m is the number of edges of the graph).

EXAMPLES:

The Petersen Graph is not weakly chordal and contains a hole:

```
sage: g = graphs.PetersenGraph()
sage: r,s = g.is_weakly_chordal(certificate=True)
sage: r
False
sage: l = s.order()
sage: s.is_isomorphic(graphs.CycleGraph(l))
True
```

join(*other*, *labels='pairs'*, *immutable=None*)

Return the join of `self` and `other`.

INPUT:

- `labels` – (defaults to 'pairs'); if set to 'pairs', each element v in the first graph will be named $(0, v)$ and each element u in `other` will be named $(1, u)$ in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.
- `immutable` – boolean (default: `None`); whether to create a mutable/immutable join. `immutable=None` (default) means that the graphs and their join will behave the same way.

See also:

- *union()*
- *disjoint_union()*

EXAMPLES:

```
sage: G = graphs.CycleGraph(3)
sage: H = Graph(2)
sage: J = G.join(H); J
Cycle graph join : Graph on 5 vertices
sage: J.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
sage: J = G.join(H, labels='integers'); J
Cycle graph join : Graph on 5 vertices
sage: J.vertices()
[0, 1, 2, 3, 4]
sage: J.edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (0, 4, None), (1, 2, None), (1, 3,␣
↪None), (1, 4, None), (2, 3, None), (2, 4, None)]
```

```
sage: G = Graph(3)
sage: G.name("Graph on 3 vertices")
sage: H = Graph(2)
sage: H.name("Graph on 2 vertices")
sage: J = G.join(H); J
Graph on 3 vertices join Graph on 2 vertices: Graph on 5 vertices
```

(continues on next page)

(continued from previous page)

```
sage: J.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
sage: J = G.join(H, labels='integers'); J
Graph on 3 vertices join Graph on 2 vertices: Graph on 5 vertices
sage: J.edges()
[(0, 3, None), (0, 4, None), (1, 3, None), (1, 4, None), (2, 3, None), (2, 4,␣
↪None)]
```

kirchhoff_symanzik_polynomial(*name='t'*)

Return the Kirchhoff-Symanzik polynomial of a graph.

This is a polynomial in variables t_e (each of them representing an edge of the graph G) defined as a sum over all spanning trees:

$$\Psi_G(t) = \sum_{\substack{T \subseteq V \\ \text{a spanning tree}}} \prod_{e \notin E(T)} t_e$$

This is also called the first Symanzik polynomial or the Kirchhoff polynomial.

INPUT:

- `name` – name of the variables (default: `'t'`)

OUTPUT:

- a polynomial with integer coefficients

ALGORITHM:

This is computed here using a determinant, as explained in Section 3.1 of [Mar2009a].

As an intermediate step, one computes a cycle basis $\mathcal{C}$ of G and a rectangular $|\mathcal{C}| \times |E(G)|$ matrix with entries in $\{-1, 0, 1\}$, which describes which edge belong to which cycle of $\mathcal{C}$ and their respective orientations.

More precisely, after fixing an arbitrary orientation for each edge $e \in E(G)$ and each cycle $C \in \mathcal{C}$, one gets a sign for every incident pair (edge, cycle) which is 1 if the orientation coincide and -1 otherwise.

EXAMPLES:

For the cycle of length 5:

```
sage: G = graphs.CycleGraph(5)
sage: G.kirchhoff_symanzik_polynomial()
t0 + t1 + t2 + t3 + t4
```

One can use another letter for variables:

```
sage: G.kirchhoff_symanzik_polynomial(name='u')
u0 + u1 + u2 + u3 + u4
```

For the 'coffee bean' graph:

```
sage: G = Graph([(0,1,'a'),(0,1,'b'),(0,1,'c')], multiedges=True)
sage: G.kirchhoff_symanzik_polynomial()
t0*t1 + t0*t2 + t1*t2
```

For the 'parachute' graph:

```
sage: G = Graph([(0,2,'a'),(0,2,'b'),(0,1,'c'),(1,2,'d')], multiedges=True)
sage: G.kirchhoff_symanzik_polynomial()
t0*t1 + t0*t2 + t1*t2 + t1*t3 + t2*t3
```

For the complete graph with 4 vertices:

```
sage: G = graphs.CompleteGraph(4)
sage: G.kirchhoff_symanzik_polynomial()
t0*t1*t3 + t0*t2*t3 + t1*t2*t3 + t0*t1*t4 + t0*t2*t4 + t1*t2*t4
+ t1*t3*t4 + t2*t3*t4 + t0*t1*t5 + t0*t2*t5 + t1*t2*t5 + t0*t3*t5
+ t2*t3*t5 + t0*t4*t5 + t1*t4*t5 + t3*t4*t5
```

REFERENCES:

[Bro2011]

least_effective_resistance(*nonedgesonly=True*)

Return a list of pairs of nodes with the least effective resistance.

The resistance distance between vertices i and j of a simple connected graph G is defined as the effective resistance between the two vertices on an electrical network constructed from G replacing each edge of the graph by a unit (1 ohm) resistor.

INPUT:

- `nonedgesonly` – Boolean (default: $True$); if true, assign zero resistance to pairs of adjacent vertices

OUTPUT: list

EXAMPLES:

Pairs of non-adjacent nodes with least effective resistance in a straight linear 2-tree on 6 vertices:

```
sage: G = Graph([(0,1),(0,2),(1,2),(1,3),(3,5),(2,4),(2,3),(3,4),(4,5)])
sage: G.least_effective_resistance()
[(1, 4)]
```

Pairs of (adjacent or non-adjacent) nodes with least effective resistance in a straight linear 2-tree on 6 vertices

```
sage: G.least_effective_resistance(nonedgesonly = False)
[(2, 3)]
```

Pairs of non-adjacent nodes with least effective resistance in a fan on 6 vertices counting only non-adjacent vertex pairs

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.least_effective_resistance()
[(2, 4)]
```

See also:

- *effective_resistance_matrix()* – a similar method giving a matrix full of all effective resistances
- *effective_resistance()* – compuetes effective resistance for a single node pair
- See Wikipedia article Resistance_distance for more details.

lex_M(*triangulation=False*, *labels=False*, *initial_vertex=None*, *algorithm=None*)

Return an ordering of the vertices according the LexM graph traversal.

LexM is a lexicographic ordering scheme that is a special type of breadth-first-search. LexM can also produce a triangulation of the given graph. This functionality is implemented in this method. For more details on the algorithms used see Sections 4 (`'lex_M_slow'`) and 5.3 (`'lex_M_fast'`) of [RTL76].

Note: This method works only for undirected graphs.

INPUT:

- `triangulation` – boolean (default: `False`); whether to return a list of edges that need to be added in order to triangulate the graph
- `labels` – boolean (default: `False`); whether to return the labels assigned to each vertex
- `initial_vertex` – (default: `None`); the first vertex to consider
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'lex_M_slow'`: slower implementation of LexM traversal
  - `'lex_M_fast'`: faster implementation of LexM traversal (works only when `labels` is set to `False`)
  - `None`: Sage chooses the best algorithm: `'lex_M_slow'` if `labels` is set to `True`, `'lex_M_fast'` otherwise.

OUTPUT:

Depending on the values of the parameters `triangulation` and `labels` the method will return one or more of the following (in that order):

- an ordering of vertices of the graph according to LexM ordering scheme
- the labels assigned to each vertex
- a list of edges that when added to the graph will triangulate it

EXAMPLES:

LexM produces an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: ord = g.lex_M(algorithm='lex_M_fast')
sage: len(ord) == g.order()
True
sage: set(ord) == set(g.vertices())
True
sage: ord = g.lex_M(algorithm='lex_M_slow')
sage: len(ord) == g.order()
True
sage: set(ord) == set(g.vertices())
True
```

Both algorithms produce a valid LexM ordering α (i.e the neighbors of $\alpha(i)$ in $G[\{\alpha(i), ..., \alpha(n)\}]$ induce a clique):

```
sage: from sage.graphs.traversals import is_valid_lex_M_order
sage: G = graphs.PetersenGraph()
sage: ord, F = G.lex_M(triangulation=True, algorithm='lex_M_slow')
```

(continues on next page)

(continued from previous page)

```
sage: is_valid_lex_M_order(G, ord, F)
True
sage: ord, F = G.lex_M(triangulation=True, algorithm='lex_M_fast')
sage: is_valid_lex_M_order(G, ord, F)
True
```

LexM produces a triangulation of given graph:

```
sage: G = graphs.PetersenGraph()
sage: _, F = G.lex_M(triangulation=True)
sage: H = Graph(F, format='list_of_edges')
sage: H.is_chordal()
True
```

LexM ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4,␣
↪5), (5, 6)])
sage: g.lex_M()
[6, 4, 5, 3, 2, 1]
```

lovasz_theta(*graph*)

Return the value of Lovász theta-function of graph

For a graph G this function is denoted by $\theta(G)$, and it can be computed in polynomial time. Mathematically, its most important property is the following:

$$\alpha(G) \leq \theta(G) \leq \chi(\overline{G})$$

with $\alpha(G)$ and $\chi(\overline{G})$ being, respectively, the maximum size of an *independent set* set of G and the *chromatic number* of the *complement* $\overline{G}$ of G.

For more information, see the Wikipedia article Lovász_number.

Note:

- Implemented for undirected graphs only. Use `to_undirected` to convert a digraph to an undirected graph.
- This function requires the optional package `csdp`, which you can install with `sage -i csdp`.

EXAMPLES:

```
sage: C = graphs.PetersenGraph()
sage: C.lovasz_theta()                                  # optional csdp
4.0
sage: graphs.CycleGraph(5).lovasz_theta()               # optional csdp
2.236068
```

magnitude_function()

Return the magnitude function of the graph as a rational function.

This is defined as the sum of all coefficients in the inverse of the matrix Z whose coefficient $Z_{i,j}$ indexed by a pair of vertices (i, j) is $q^d(i, j)$ where d is the distance function in the graph.

By convention, if the distance from i to j is infinite (for two vertices not path connected) then $Z_{i,j} = 0$.

The value of the magnitude function at $q = 0$ is the cardinality of the graph. The magnitude function of a disjoint union is the sum of the magnitudes functions of the connected components. The magnitude function of a Cartesian product is the product of the magnitudes functions of the factors.

EXAMPLES:

```
sage: g = Graph({1:[], 2:[]})
sage: g.magnitude_function()
2

sage: g = graphs.CycleGraph(4)
sage: g.magnitude_function()
4/(q^2 + 2*q + 1)

sage: g = graphs.CycleGraph(5)
sage: m = g.magnitude_function(); m
5/(2*q^2 + 2*q + 1)
```

One can expand the magnitude as a power series in q as follows:

```
sage: q = QQ[['q']].gen()
sage: m(q)
5 - 10*q + 10*q^2 - 20*q^4 + 40*q^5 - 40*q^6 + ...
```

One can also use the substitution $q = exp(-t)$ to obtain the magnitude function as a function of t:

```
sage: g = graphs.CycleGraph(6)
sage: m = g.magnitude_function()
sage: t = var('t')
sage: m(exp(-t))
6/(2*e^(-t) + 2*e^(-2*t) + e^(-3*t) + 1)
```

REFERENCES:

matching(*value_only=False*, *algorithm='Edmonds'*, *use_edge_labels=False*, *solver=None*, *verbose=0*)

Return a maximum weighted matching of the graph represented by the list of its edges.

For more information, see the Wikipedia article Matching_(graph_theory).

Given a graph G such that each edge e has a weight w_e, a maximum matching is a subset S of the edges of G of maximum weight such that no two edges of S are incident with each other.

As an optimization problem, it can be expressed as:

$$\text{Maximize : } \sum_{e \in G.edges()} w_e b_e$$

$$\text{Such that : } \forall v \in G, \sum_{(u,v) \in G.edges()} b_{(u,v)} \leq 1$$

$$\forall x \in G, b_x \text{ is a binary variable}$$

INPUT:

- `value_only` – boolean (default: `False`); when set to `True`, only the cardinal (or the weight) of the matching is returned
- `algorithm` – string (default: `"Edmonds"`)
  - `"Edmonds"` selects Edmonds' algorithm as implemented in NetworkX
  - `"LP"` uses a Linear Program formulation of the matching problem

- use_edge_labels – boolean (default: False)
  - when set to True, computes a weighted matching where each edge is weighted by its label (if an edge has no label, 1 is assumed)
  - when set to False, each edge has weight 1
- solver – (default: None); specify a Linear Program (LP) solver to be used; if set to None, the default one is used
- verbose – integer (default: 0); sets the level of verbosity: set to 0 by default, which means quiet (only useful when algorithm == "LP")

For more information on LP solvers and which default solver is used, see the method sage.numerical.mip.MixedIntegerLinearProgram.solve() of the class sage.numerical.mip.MixedIntegerLinearProgram.

OUTPUT:

- When value_only=False (default), this method returns the list of edges of a maximum matching of G.
- When value_only=True, this method returns the sum of the weights (default: 1) of the edges of a maximum matching of G. The type of the output may vary according to the type of the edge labels and the algorithm used.

ALGORITHM:

The problem is solved using Edmond's algorithm implemented in NetworkX, or using Linear Programming depending on the value of algorithm.

EXAMPLES:

Maximum matching in a Pappus Graph:

```
sage: g = graphs.PappusGraph()
sage: g.matching(value_only=True)
9
```

Same test with the Linear Program formulation:

```
sage: g = graphs.PappusGraph()
sage: g.matching(algorithm="LP", value_only=True)
9
```

matching_polynomial(*G*, *complement=True*, *name=None*)

Computes the matching polynomial of the graph G.

If $p(G, k)$ denotes the number of k-matchings (matchings with k edges) in G, then the matching polynomial is defined as [God1993]:

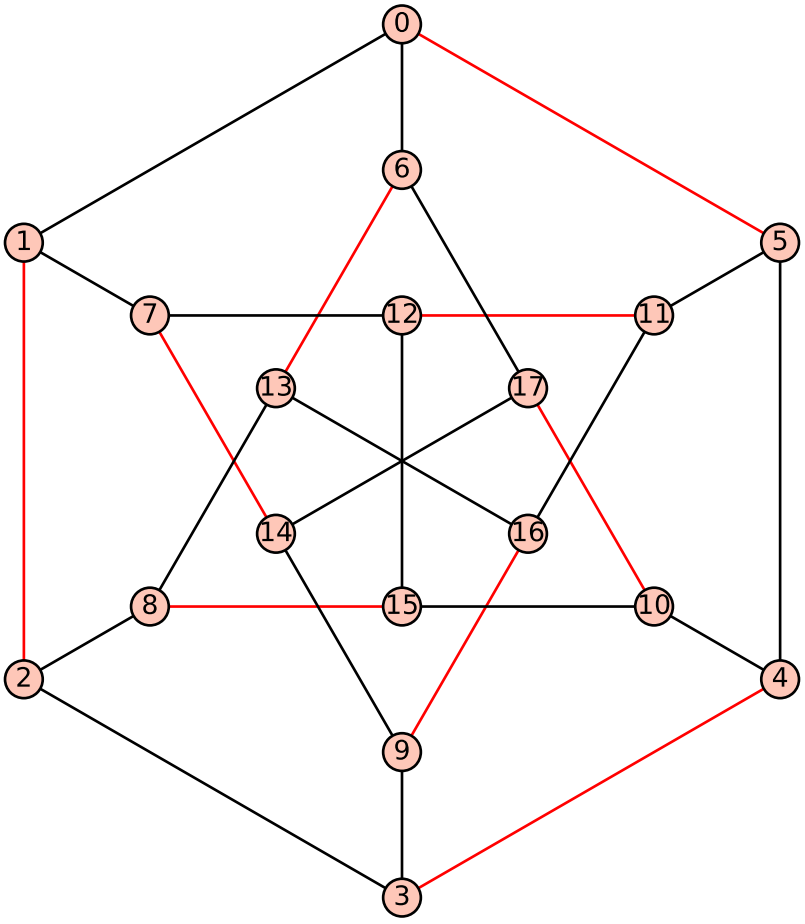
$$\mu(x) = \sum_{k \geq 0} (-1)^k p(G, k) x^{n-2k}$$

INPUT:

- complement - (default: True) whether to use Godsil's duality theorem to compute the matching polynomial from that of the graphs complement (see ALGORITHM).
- name - optional string for the variable name in the polynomial

---

Note: The complement option uses matching polynomials of complete graphs, which are cached. So if you are crazy enough to try computing the matching polynomial on a graph with millions of vertices,


0
6
1
5
7
12
11
13
17
14
16
8
15
10
2
4
9
3

you might not want to use this option, since it will end up caching millions of polynomials of degree in the millions.

ALGORITHM:

The algorithm used is a recursive one, based on the following observation [God1993]:

- If e is an edge of G, G' is the result of deleting the edge e, and G'' is the result of deleting each vertex in e, then the matching polynomial of G is equal to that of G' minus that of G''.

  (the algorithm actually computes the *signless* matching polynomial, for which the recursion is the same when one replaces the substraction by an addition. It is then converted into the matching polynomial and returned)

Depending on the value of `complement`, Godsil's duality theorem [God1993] can also be used to compute $\mu(x)$:

$$\mu(\overline{G}, x) = \sum_{k \geq 0} p(G, k)\mu(K_{n-2k}, x)$$

Where $\overline{G}$ is the complement of G, and K_n the complete graph on n vertices.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.matching_polynomial()
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(complement=False)
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(name='tom')
tom^10 - 15*tom^8 + 75*tom^6 - 145*tom^4 + 90*tom^2 - 6
sage: g = Graph()
sage: L = [graphs.RandomGNP(8, .3) for i in range(1, 6)]
sage: prod([h.matching_polynomial() for h in L]) == sum(L, g).matching_
↪polynomial()  # long time (up to 10s on sage.math, 2011)
True
```

```
sage: for i in range(1, 12):  # long time (10s on sage.math, 2011)
....:     for t in graphs.trees(i):
....:         if t.matching_polynomial() != t.characteristic_polynomial():
....:             raise RuntimeError('bug for a tree A of size {0}'.format(i))
....:         c = t.complement()
....:         if c.matching_polynomial(complement=False) != c.matching_
↪polynomial():
....:             raise RuntimeError('bug for a tree B of size {0}'.format(i))
```

```
sage: from sage.graphs.matchpoly import matching_polynomial
sage: matching_polynomial(graphs.CompleteGraph(0))
1
sage: matching_polynomial(graphs.CompleteGraph(1))
x
sage: matching_polynomial(graphs.CompleteGraph(2))
x^2 - 1
sage: matching_polynomial(graphs.CompleteGraph(3))
x^3 - 3*x
sage: matching_polynomial(graphs.CompleteGraph(4))
x^4 - 6*x^2 + 3
sage: matching_polynomial(graphs.CompleteGraph(5))
```

(continues on next page)

(continued from previous page)

```
x^5 - 10*x^3 + 15*x
sage: matching_polynomial(graphs.CompleteGraph(6))
x^6 - 15*x^4 + 45*x^2 - 15
sage: matching_polynomial(graphs.CompleteGraph(7))
x^7 - 21*x^5 + 105*x^3 - 105*x
sage: matching_polynomial(graphs.CompleteGraph(8))
x^8 - 28*x^6 + 210*x^4 - 420*x^2 + 105
sage: matching_polynomial(graphs.CompleteGraph(9))
x^9 - 36*x^7 + 378*x^5 - 1260*x^3 + 945*x
sage: matching_polynomial(graphs.CompleteGraph(10))
x^10 - 45*x^8 + 630*x^6 - 3150*x^4 + 4725*x^2 - 945
sage: matching_polynomial(graphs.CompleteGraph(11))
x^11 - 55*x^9 + 990*x^7 - 6930*x^5 + 17325*x^3 - 10395*x
sage: matching_polynomial(graphs.CompleteGraph(12))
x^12 - 66*x^10 + 1485*x^8 - 13860*x^6 + 51975*x^4 - 62370*x^2 + 10395
sage: matching_polynomial(graphs.CompleteGraph(13))
x^13 - 78*x^11 + 2145*x^9 - 25740*x^7 + 135135*x^5 - 270270*x^3 + 135135*x
```

```
sage: G = Graph({0:[1,2], 1:[2]})
sage: matching_polynomial(G)
x^3 - 3*x
sage: G = Graph({0:[1,2]})
sage: matching_polynomial(G)
x^3 - 2*x
sage: G = Graph({0:[1], 2:[]})
sage: matching_polynomial(G)
x^3 - x
sage: G = Graph({0:[], 1:[], 2:[]})
sage: matching_polynomial(G)
x^3
```

```
sage: matching_polynomial(graphs.CompleteGraph(0), complement=False)
1
sage: matching_polynomial(graphs.CompleteGraph(1), complement=False)
x
sage: matching_polynomial(graphs.CompleteGraph(2), complement=False)
x^2 - 1
sage: matching_polynomial(graphs.CompleteGraph(3), complement=False)
x^3 - 3*x
sage: matching_polynomial(graphs.CompleteGraph(4), complement=False)
x^4 - 6*x^2 + 3
sage: matching_polynomial(graphs.CompleteGraph(5), complement=False)
x^5 - 10*x^3 + 15*x
sage: matching_polynomial(graphs.CompleteGraph(6), complement=False)
x^6 - 15*x^4 + 45*x^2 - 15
sage: matching_polynomial(graphs.CompleteGraph(7), complement=False)
x^7 - 21*x^5 + 105*x^3 - 105*x
sage: matching_polynomial(graphs.CompleteGraph(8), complement=False)
x^8 - 28*x^6 + 210*x^4 - 420*x^2 + 105
sage: matching_polynomial(graphs.CompleteGraph(9), complement=False)
x^9 - 36*x^7 + 378*x^5 - 1260*x^3 + 945*x
sage: matching_polynomial(graphs.CompleteGraph(10), complement=False)
x^10 - 45*x^8 + 630*x^6 - 3150*x^4 + 4725*x^2 - 945
sage: matching_polynomial(graphs.CompleteGraph(11), complement=False)
x^11 - 55*x^9 + 990*x^7 - 6930*x^5 + 17325*x^3 - 10395*x
sage: matching_polynomial(graphs.CompleteGraph(12), complement=False)
```

(continues on next page)

(continued from previous page)

```
x^12 - 66*x^10 + 1485*x^8 - 13860*x^6 + 51975*x^4 - 62370*x^2 + 10395
sage: matching_polynomial(graphs.CompleteGraph(13), complement=False)
x^13 - 78*x^11 + 2145*x^9 - 25740*x^7 + 135135*x^5 - 270270*x^3 + 135135*x
```

maximum_average_degree(*value_only=True*, *solver=None*, *verbose=0*)

Return the Maximum Average Degree (MAD) of the current graph.

The Maximum Average Degree (MAD) of a graph is defined as the average degree of its densest subgraph. More formally, `Mad(G) = \max_{H\subseteq G} Ad(H)`, where $Ad(G)$ denotes the average degree of G.

This can be computed in polynomial time.

INPUT:

- `value_only` – boolean (default: `True`);
  - If `value_only=True`, only the numerical value of the MAD is returned.
  - Else, the subgraph of G realizing the MAD is returned.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

In any graph, the Mad is always larger than the average degree:

```
sage: g = graphs.RandomGNP(20,.3)
sage: mad_g = g.maximum_average_degree()
sage: g.average_degree() <= mad_g
True
```

Unlike the average degree, the Mad of the disjoint union of two graphs is the maximum of the Mad of each graphs:

```
sage: h = graphs.RandomGNP(20,.3)
sage: mad_h = h.maximum_average_degree()
sage: (g+h).maximum_average_degree() == max(mad_g, mad_h)
True
```

The subgraph of a regular graph realizing the maximum average degree is always the whole graph

```
sage: g = graphs.CompleteGraph(5)
sage: mad_g = g.maximum_average_degree(value_only=False)
sage: g.is_isomorphic(mad_g)
True
```

This also works for complete bipartite graphs

```
sage: g = graphs.CompleteBipartiteGraph(3,4)
sage: mad_g = g.maximum_average_degree(value_only=False)
sage: g.is_isomorphic(mad_g)
True
```

maximum_cardinality_search(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Return an ordering of the vertices according a maximum cardinality search.

Maximum cardinality search (MCS) is a graph traversal introduced in [TY1984]. It starts by assigning an arbitrary vertex (or the specified `initial_vertex`) of G the last position in the ordering α. Every vertex keeps a weight equal to the number of its already processed neighbors (i.e., already added to α), and a vertex of largest such number is chosen at each step i to be placed in position $n - i$ in α. This ordering can be computed in time $O(n + m)$.

When the graph is chordal, the ordering returned by MCS is a *perfect elimination ordering*, like *lex_BFS()*. So this ordering can be used to recognize chordal graphs. See [He2006] for more details.

Note: The current implementation is for connected graphs only.

INPUT:

- `G` – a Sage Graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to also return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

OUTPUT:

By default, return the ordering α as a list. When `tree` is `True`, the method returns a tuple (α, T), where T is a directed tree with the same set of vertices as G to v if u was the first vertex to saw v.

EXAMPLES:

When specified, the `initial_vertex` is placed at the end of the ordering, unless parameter `reverse` is `True`, in which case it is placed at the beginning:

```
sage: G = graphs.PathGraph(4)
sage: G.maximum_cardinality_search(initial_vertex=0)
[3, 2, 1, 0]
sage: G.maximum_cardinality_search(initial_vertex=1)
[0, 3, 2, 1]
sage: G.maximum_cardinality_search(initial_vertex=2)
[0, 1, 3, 2]
sage: G.maximum_cardinality_search(initial_vertex=3)
[0, 1, 2, 3]
sage: G.maximum_cardinality_search(initial_vertex=3, reverse=True)
[3, 2, 1, 0]
```

Returning the discovery tree:

```
sage: G = graphs.PathGraph(4)
sage: _, T = G.maximum_cardinality_search(tree=True, initial_vertex=0)
sage: T.order(), T.size()
(4, 3)
sage: T.edges(labels=False, sort=True)
[(1, 0), (2, 1), (3, 2)]
sage: _, T = G.maximum_cardinality_search(tree=True, initial_vertex=3)
sage: T.edges(labels=False, sort=True)
[(0, 1), (1, 2), (2, 3)]
```

maximum_cardinality_search_M(*G*, *initial_vertex=None*)

Return the ordering and the edges of the triangulation produced by MCS-M.

Maximum cardinality search M (MCS-M) is an extension of MCS (*maximum_cardinality_search()*) in the same way that Lex-M (*lex_M()*) is an extension of Lex-BFS (*lex_BFS()*). That is, in MCS-M when u receives number i at step $n - i + 1$, it increments the weight of all unnumbered vertices v for which there exists a path between u and v consisting only of unnumbered vertices with weight strictly less than $w^-(u)$ and $w^-(v)$, where w^- is the number of times a vertex has been reached during previous iterations. See [BBHP2004] for the details of this $O(nm)$ time algorithm.

If G is not connected, the orderings of each of its connected components are added consecutively. Furthermore, if G has k connected components C_i for $0 \leq i < k$, X contains at least one vertex of C_i for each $i \geq 1$. Hence, $|X| \geq k - 1$. In particular, some isolated vertices (i.e., of degree 0) can appear in X as for such a vertex x, we have that $G \setminus N(x) = G$ is not connected.

INPUT:

- `G` – a Sage graph
- `initial_vertex` – (default: `None`); the first vertex to consider

OUTPUT: a tuple (α, F, X), where

- α is the resulting ordering of the vertices. If an initial vertex is specified, it gets the last position in the ordering α.
- F is the list of edges of a minimal triangulation of G according α
- X is is a list of vertices such that for each $x \in X$, the neighborhood of x in G is a separator (i.e., $G \setminus N(x)$ is not connected). Note that we may have $N(x) = \emptyset$ if G is not connected and x has degree 0.

EXAMPLES:

Chordal graphs have a perfect elimination ordering, and so the set F of edges of the triangulation is empty:

```
sage: G = graphs.RandomChordalGraph(20)
sage: alpha, F, X = G.maximum_cardinality_search_M(); F
[]
```

The cycle of order 4 is not chordal and so the triangulation has one edge:

```
sage: G = graphs.CycleGraph(4)
sage: alpha, F, X = G.maximum_cardinality_search_M(); len(F)
1
```

The number of edges needed to triangulate of a cycle graph or order n is $n-3$, independently of the initial vertex:

```
sage: n = randint(3, 20)
sage: C = graphs.CycleGraph(n)
sage: _, F, X = C.maximum_cardinality_search_M()
sage: len(F) == n - 3
True
sage: _, F, X = C.maximum_cardinality_search_M(initial_vertex=C.random_
↪vertex())
sage: len(F) == n - 3
True
```

When an initial vertex is specified, it gets the last position in the ordering:

```
sage: G = graphs.PathGraph(4)
sage: G.maximum_cardinality_search_M(initial_vertex=0)
([3, 2, 1, 0], [], [2, 3])
sage: G.maximum_cardinality_search_M(initial_vertex=1)
([3, 2, 0, 1], [], [2, 3])
sage: G.maximum_cardinality_search_M(initial_vertex=2)
([0, 1, 3, 2], [], [0, 1])
sage: G.maximum_cardinality_search_M(initial_vertex=3)
([0, 1, 2, 3], [], [0, 1])
```

When G is not connected, the orderings of each of its connected components are added consecutively, the vertices of the component containing the initial vertex occupying the last positions:

```
sage: G = graphs.CycleGraph(4) * 2
sage: G.maximum_cardinality_search_M()[0]
[5, 4, 6, 7, 2, 3, 1, 0]
sage: G.maximum_cardinality_search_M(initial_vertex=7)[0]
[2, 1, 3, 0, 5, 6, 4, 7]
```

Furthermore, if G has k connected components, X contains at least one vertex per connected component, except for the first one, and so at least $k-1$ vertices:

```
sage: for k in range(1, 5):
....:     _, _, X = Graph(k).maximum_cardinality_search_M()
....:     if len(X) < k - 1:
....:         raise ValueError("something goes wrong")
sage: G = graphs.RandomGNP(10, .2)
sage: cc = G.connected_components()
sage: _, _, X = G.maximum_cardinality_search_M()
sage: len(X) >= len(cc) - 1
True
```

In the example of [BPS2010], the triangulation has 3 edges:

```
sage: G = Graph({'a': ['b', 'k'], 'b': ['c'], 'c': ['d', 'j', 'k'],
....:            'd': ['e', 'f', 'j', 'k'], 'e': ['g'],
....:            'f': ['g', 'j', 'k'], 'g': ['j', 'k'], 'h': ['i', 'j'],
....:            'i': ['k'], 'j': ['k']})
sage: _, F, _ = G.maximum_cardinality_search_M(initial_vertex='a')
sage: len(F)
3
```

minimal_dominating_sets(*G*, *to_dominate=None*, *work_on_copy=True*)

Return an iterator over the minimal dominating sets of a graph.

INPUT:

- `G` – a graph.
- `to_dominate` – vertex iterable or `None` (default: `None`); the set of vertices to be dominated.
- `work_on_copy` – boolean (default: `True`); whether or not to work on a copy of the input graph; if set to `False`, the input graph will be modified (relabeled).

OUTPUT:

An iterator over the inclusion-minimal sets of vertices of `G`. If `to_dominate` is provided, return an iterator over the inclusion-minimal sets of vertices that dominate the vertices of `to_dominate`.

ALGORITHM: The algorithm described in [BDHPR2019].

AUTHOR: Jean-Florent Raymond (2019-03-04) – initial version.

EXAMPLES:

```
sage: G = graphs.ButterflyGraph()
sage: ll = list(G.minimal_dominating_sets())
sage: pp = [{0, 1}, {1, 3}, {0, 2}, {2, 3}, {4}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x␣
↪in pp)
True

sage: ll = list(G.minimal_dominating_sets([0,3]))
sage: pp = [{0}, {3}, {4}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x␣
↪in pp)
True

sage: ll = list(G.minimal_dominating_sets([4]))
sage: pp = [{4}, {0}, {1}, {2}, {3}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x␣
↪in pp)
True
```

```
sage: ll = list(graphs.PetersenGraph().minimal_dominating_sets())
sage: pp = [{0, 2, 6},
....: {0, 9, 3},
....: {0, 8, 7},
....: {1, 3, 7},
....: {1, 4, 5},
....: {8, 1, 9},
....: {8, 2, 4},
....: {9, 2, 5},
....: {3, 5, 6},
....: {4, 6, 7},
....: {0, 8, 2, 9},
....: {0, 3, 6, 7},
....: {1, 3, 5, 9},
....: {8, 1, 4, 7},
....: {2, 4, 5, 6},
....: {0, 1, 2, 3, 4},
....: {0, 1, 2, 5, 7},
....: {0, 1, 4, 6, 9},
....: {0, 1, 5, 6, 8},
....: {0, 8, 3, 4, 5},
....: {0, 9, 4, 5, 7},
....: {8, 1, 2, 3, 6},
....: {1, 2, 9, 6, 7},
....: {9, 2, 3, 4, 7},
....: {8, 2, 3, 5, 7},
....: {8, 9, 3, 4, 6},
....: {8, 9, 5, 6, 7}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x␣
↪in pp)
True
```

minimum_outdegree_orientation(*use_edge_labels=False*, *solver=None*, *verbose=0*)

Returns an orientation of `self` with the smallest possible maximum outdegree.

Given a Graph G, it is polynomial to compute an orientation D of the edges of G such that the maximum

out-degree in D is minimized. This problem, though, is NP-complete in the weighted case [AMOZ2006].

INPUT:

- `use_edge_labels` – boolean (default: `False`)
  - When set to `True`, uses edge labels as weights to compute the orientation and assumes a weight of 1 when there is no value available for a given edge.
  - When set to `False` (default), gives a weight of 1 to all the edges.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

Given a complete bipartite graph $K_{n,m}$, the maximum out-degree of an optimal orientation is $\left\lceil \frac{nm}{n+m} \right\rceil$:

```
sage: g = graphs.CompleteBipartiteGraph(3,4)
sage: o = g.minimum_outdegree_orientation()
sage: max(o.out_degree()) == ceil((4*3)/(3+4))
True
```

minor(*H*, *solver=None*, *verbose=0*)

Return the vertices of a minor isomorphic to H in the current graph.

We say that a graph G has a H-minor (or that it has a graph isomorphic to H as a minor), if for all $h \in H$, there exist disjoint sets $S_h \subseteq V(G)$ such that once the vertices of each S_h have been merged to create a new graph G', this new graph contains H as a subgraph.

For more information, see the Wikipedia article Minor_(graph_theory).

INPUT:

- `H` – The minor to find for in the current graph.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

A dictionary associating to each vertex of H the set of vertices in the current graph representing it.

ALGORITHM:

Mixed Integer Linear Programming

COMPLEXITY:

Theoretically, when H is fixed, testing for the existence of a H-minor is polynomial. The known algorithms are highly exponential in H, though.

Note: This function can be expected to be *very* slow, especially where the minor does not exist.

EXAMPLES:

Trying to find a minor isomorphic to K_4 in the 4×4 grid:

```
sage: g = graphs.GridGraph([4,4])
sage: h = graphs.CompleteGraph(4)
sage: L = g.minor(h)
sage: gg = g.subgraph(flatten(L.values(), max_level = 1))
sage: _ = [gg.merge_vertices(l) for l in L.values() if len(l)>1]
sage: gg.is_isomorphic(h)
True
```

We can also try to prove this way that the Petersen graph is not planar, as it has a K_5 minor:

```
sage: g = graphs.PetersenGraph()
sage: K5_minor = g.minor(graphs.CompleteGraph(5))                    # long␣
↪time
```

And even a $K_{3,3}$ minor:

```
sage: K33_minor = g.minor(graphs.CompleteBipartiteGraph(3,3))        # long␣
↪time
```

(It is much faster to use the linear-time test of planarity in this situation, though.)

As there is no cycle in a tree, looking for a K_3 minor is useless. This function will raise an exception in this case:

```
sage: g = graphs.RandomGNP(20,.5)
sage: g = g.subgraph(edges = g.min_spanning_tree())
sage: g.is_tree()
True
sage: L = g.minor(graphs.CompleteGraph(3))
Traceback (most recent call last):
...
ValueError: This graph has no minor isomorphic to H !
```

modular_decomposition (*algorithm='habib'*, *style='tuple'*)

Return the modular decomposition of the current graph.

A module of an undirected graph is a subset of vertices such that every vertex outside the module is either connected to all members of the module or to none of them. Every graph that has a nontrivial module can be partitioned into modules, and the increasingly fine partitions into modules form a tree. The `modular_decomposition` function returns that tree.

INPUT:

- `algorithm` – string (default: `'habib'`); specifies the algorithm to use among:
  - `'tedder'` – linear time algorithm of [TCHP2008]
  - `'habib'` – $O(n^3)$ algorithm of [HM1979]. This algorithm is much simpler and so possibly less prone to errors.
- `style` – string (default: `'tuple'`); specifies the output format:
  - `'tuple'` – as nested tuples.
  - `'tree'` – as `LabelledRootedTree`.

OUTPUT:

A pair of two values (recursively encoding the decomposition) :

- The type of the current module :

  - `"PARALLEL"`
  - `"PRIME"`
  - `"SERIES"`

- The list of submodules (as list of pairs `(type, list)`, recursively...) or the vertex's name if the module is a singleton.

Crash course on modular decomposition:

A module M of a graph G is a proper subset of its vertices such that for all $u \in V(G) - M, v, w \in M$ the relation $u \sim v \Leftrightarrow u \sim w$ holds, where $\sim$ denotes the adjacency relation in G. Equivalently, $M \subset V(G)$ is a module if all its vertices have the same adjacency relations with each vertex outside of the module (vertex by vertex).

Hence, for a set like a module, it is very easy to encode the information of the adjacencies between the vertices inside and outside the module – we can actually add a new vertex v_M to our graph representing our module M, and let v_M be adjacent to $u \in V(G) - M$ if and only if some $v \in M$ (and hence all the vertices contained in the module) is adjacent to u. We can now independently (and recursively) study the structure of our module M and the new graph $G - M + \{v_M\}$, without any loss of information.

Here are two very simple modules :

- A connected component C (or the union of some –but not all– of them) of a disconnected graph G, for instance, is a module, as no vertex of C has a neighbor outside of it.
- An anticomponent C (or the union of some –but not all– of them) of an non-anticonnected graph G, for the same reason (it is just the complement of the previous graph !).

These modules being of special interest, the disjoint union of graphs is called a Parallel composition, and the complement of a disjoint union is called a Series composition. A graph whose only modules are singletons is called Prime.

For more information on modular decomposition, in particular for an explanation of the terms "Parallel," "Prime" and "Series," see the Wikipedia article Modular_decomposition.

You may also be interested in the survey from Michel Habib and Christophe Paul entitled "A survey on Algorithmic aspects of modular decomposition" [HP2010].

EXAMPLES:

The Bull Graph is prime:

```
sage: graphs.BullGraph().modular_decomposition()  # py2
(PRIME, [0, 3, 4, 2, 1])
sage: graphs.BullGraph().modular_decomposition()  # py3
(PRIME, [1, 2, 0, 3, 4])
```

The Petersen Graph too:

```
sage: graphs.PetersenGraph().modular_decomposition()  # py2
(PRIME, [6, 2, 5, 1, 9, 3, 0, 7, 8, 4])
sage: graphs.PetersenGraph().modular_decomposition()  # py3
(PRIME, [1, 4, 5, 0, 2, 6, 3, 7, 8, 9])
```

This a clique on 5 vertices with 2 pendant edges, though, has a more interesting decomposition:

```
sage: g = graphs.CompleteGraph(5)
sage: g.add_edge(0,5)
sage: g.add_edge(0,6)
```

(continues on next page)

(continued from previous page)

```
sage: g.modular_decomposition(algorithm='habib')
(SERIES, [(PARALLEL, [(SERIES, [1, 2, 3, 4]), 5, 6]), 0])
```

We get an equivalent tree when we use the algorithm of [TCHP2008]:

```
sage: g.modular_decomposition(algorithm='tedder')
(SERIES, [(PARALLEL, [(SERIES, [4, 3, 2, 1]), 5, 6]), 0])
```

We can choose output to be a LabelledRootedTree:

```
sage: g.modular_decomposition(style='tree')
SERIES[0[], PARALLEL[5[], 6[], SERIES[1[], 2[], 3[], 4[]]]]
sage: ascii_art(g.modular_decomposition(style='tree'))
  __SERIES
 /       /
0   ___PARALLEL
   / /     /
  5 6   __SERIES
       / / / /
      1 2 3 4
```

ALGORITHM:

When `algorithm='tedder'` this function uses python implementation of algorithm published by Marc Tedder, Derek Corneil, Michel Habib and Christophe Paul [TCHP2008]. When `algorithm='habib'` this function uses the algorithm of M. Habib and M. Maurer [HM1979].

See also:

- *is_prime()* – Tests whether a graph is prime.
- LabelledRootedTree.

most_common_neighbors(*nonedgesonly=True*)

Return vertex pairs with maximal number of common neighbors.

This method is only valid for simple (no loops, no multiple edges) graphs with order ≥ 2

INPUT:

- `nonedgesonly`– boolean (default: `True`); if `True`, assigns 0 value to adjacent vertices.

OUTPUT: list of tuples of edge pairs

EXAMPLES:

The maximum common neighbor (non-adjacent) pairs for a straight linear 2-tree

```
sage: G1 = Graph([(0,1),(0,2),(1,2),(1,3),(3,5),(2,4),(2,3),(3,4),(4,5)])
sage: G1.most_common_neighbors()
[(0, 3), (1, 4), (2, 5)]
```

If we include non-adjacent pairs

```
sage: G1.most_common_neighbors(nonedgesonly = False)
[(0, 3), (1, 2), (1, 4), (2, 3), (2, 5), (3, 4)]
```

The common neighbors matrix for a fan on 6 vertices counting only non-adjacent vertex pairs

```
sage: H = Graph([(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(1,2),(2,3),(3,4),(4,5)])
sage: H.most_common_neighbors()
[(1, 3), (2, 4), (3, 5)]
```

See also:

- *common_neighbors_matrix()* – a similar method giving a matrix of number of common neighbors

orientations(*data_structure=None*, *sparse=None*)

Return an iterator over orientations of `self`.

An *orientation* of an undirected graph is a directed graph such that every edge is assigned a direction. Hence there are 2^s oriented digraphs for a simple graph with s edges.

INPUT:

- `data_structure` – one of `"sparse"`, `"static_sparse"`, or `"dense"`; see the documentation of *Graph* or *DiGraph*; default is the data structure of `self`
- `sparse` – boolean (default: `None`); `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`. By default (`None`), guess the most suitable data structure.

Warning: This always considers multiple edges of graphs as distinguishable, and hence, may have repeated digraphs.

EXAMPLES:

```
sage: G = Graph([[1,2,3], [(1, 2, 'a'), (1, 3, 'b')]], format='vertices_and_
↪edges')
sage: it = G.orientations()
sage: D = next(it)
sage: D.edges()
[(1, 2, 'a'), (1, 3, 'b')]
sage: D = next(it)
sage: D.edges()
[(1, 2, 'a'), (3, 1, 'b')]
```

pathwidth(*k=None*, *certificate=False*, *algorithm='BAB'*, *verbose=False*, *max_prefix_length=20*, *max_prefix_number=1000000*)

Compute the pathwidth of `self` (and provides a decomposition)

INPUT:

- `k` – integer (default: `None`); the width to be considered. When `k` is an integer, the method checks that the graph has pathwidth $\leq k$. If `k` is `None` (default), the method computes the optimal pathwidth.
- `certificate` – boolean (default: `False`); whether to return the path-decomposition itself
- `algorithm` – string (default: `"BAB"`); algorithm to use among:
  - `"BAB"` – Use a branch-and-bound algorithm. This algorithm has no size restriction but could take a very long time on large graphs. It can also be used to test is the input graph has pathwidth $\leq k$, in which cas it will return the first found solution with width $\leq k$ is `certificate==True`.
  - `exponential` – Use an exponential time and space algorithm. This algorithm only works of graphs on less than 32 vertices.

  - `MILP` – Use a mixed integer linear programming formulation. This algorithm has no size restriction but could take a very long time.

- `verbose` – boolean (default: `False`); whether to display information on the computations
- `max_prefix_length` – integer (default: 20); limits the length of the stored prefixes to prevent storing too many prefixes. This parameter is used only when `algorithm=="BAB"`.
- `max_prefix_number` – integer (default: 10**6); upper bound on the number of stored prefixes used to prevent using too much memory. This parameter is used only when `algorithm=="BAB"`.

OUTPUT:

Return the pathwidth of `self`. When `k` is specified, it returns `False` when no path-decomposition of width $\leq k$ exists or `True` otherwise. When `certificate=True`, the path-decomposition is also returned.

See also:

- *Graph.treewidth()* – computes the treewidth of a graph
- *vertex_separation()* – computes the vertex separation of a (di)graph

EXAMPLES:

The pathwidth of a cycle is equal to 2:

```
sage: g = graphs.CycleGraph(6)
sage: g.pathwidth()
2
sage: pw, decomp = g.pathwidth(certificate=True)
sage: sorted(decomp, key=str)
[{0, 1, 5}, {1, 2, 5}, {2, 3, 4}, {2, 4, 5}]
```

The pathwidth of a Petersen graph is 5:

```
sage: g = graphs.PetersenGraph()
sage: g.pathwidth()
5
sage: g.pathwidth(k=2)
False
sage: g.pathwidth(k=6)
True
sage: g.pathwidth(k=6, certificate=True)
(True, Graph on 5 vertices)
```

perfect_matchings(*labels=False*)

Return an iterator over all perfect matchings of the graph.

ALGORITHM:

Choose a vertex v, then recurse through all edges incident to v, removing one edge at a time whenever an edge is added to a matching.

INPUT:

- `labels` – boolean (default: `False`); when `True`, the edges in each perfect matching are triples (containing the label as the third element), otherwise the edges are pairs.

See also:

matching()

EXAMPLES:

```
sage: G=graphs.GridGraph([2,3])
sage: for m in G.perfect_matchings():
....:     print(sorted(m))
[((0, 0), (0, 1)), ((0, 2), (1, 2)), ((1, 0), (1, 1))]
[((0, 0), (1, 0)), ((0, 1), (0, 2)), ((1, 1), (1, 2))]
[((0, 0), (1, 0)), ((0, 1), (1, 1)), ((0, 2), (1, 2))]

sage: G = graphs.CompleteGraph(4)
sage: for m in G.perfect_matchings(labels=True):
....:     print(sorted(m))
[(0, 1, None), (2, 3, None)]
[(0, 2, None), (1, 3, None)]
[(0, 3, None), (1, 2, None)]

sage: G = Graph([[1,-1,'a'], [2,-2, 'b'], [1,-2,'x'], [2,-1,'y']])
sage: sorted(sorted(m) for m in G.perfect_matchings(labels=True))
[[(-2, 1, 'x'), (-1, 2, 'y')], [(-2, 2, 'b'), (-1, 1, 'a')]]

sage: G = graphs.CompleteGraph(8)
sage: mpc = G.matching_polynomial().coefficients(sparse=False)[0]
sage: len(list(G.perfect_matchings())) == mpc
True

sage: G = graphs.PetersenGraph().copy(immutable=True)
sage: list(G.perfect_matchings())
[[(0, 1), (2, 3), (4, 9), (5, 7), (6, 8)],
 [(0, 1), (2, 7), (3, 4), (5, 8), (6, 9)],
 [(0, 4), (1, 2), (3, 8), (5, 7), (6, 9)],
 [(0, 4), (1, 6), (2, 3), (5, 8), (7, 9)],
 [(0, 5), (1, 2), (3, 4), (6, 8), (7, 9)],
 [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]]

sage: list(Graph().perfect_matchings())
[[]]

sage: G = graphs.CompleteGraph(5)
sage: list(G.perfect_matchings())
[]
```

private_neighbors(*G*, *vertex*, *dom*)

Return the private neighbors of a vertex with repect to other vertices.

A private neighbor of a vertex v with respect to a vertex subset D is a closed neighbor of v that is not dominated by a vertex of $D \setminus \{v\}$.

INPUT:

- `vertex` – a vertex of `G`.
- `dom` – iterable of vertices of `G`; the vertices possibly stealing private neighbors from `vertex`.

OUTPUT:

Return the closed neighbors of `vertex` that are not closed neighbors of any other vertex of `dom`.

EXAMPLES:

```
sage: g = graphs.PathGraph(5)
sage: list(g.private_neighbors(1, [1, 3, 4]))
[1, 0]

sage: list(g.private_neighbors(1, [3, 4]))
[1, 0]

sage: list(g.private_neighbors(1, [3, 4, 0]))
[]
```

random_orientation(*G*)

Return a random orientation of a graph G.

An *orientation* of an undirected graph is a directed graph such that every edge is assigned a direction. Hence there are 2^m oriented digraphs for a simple graph with m edges.

INPUT:

- `G` – a Graph.

EXAMPLES:

```
sage: from sage.graphs.orientations import random_orientation
sage: G = graphs.PetersenGraph()
sage: D = random_orientation(G)
sage: D.order() == G.order(), D.size() == G.size()
(True, True)
```

See also:

- *orientations()*

random_spanning_tree(*output_as_graph=False*)

Return a random spanning tree of the graph.

This uses the Aldous-Broder algorithm ([Bro1989], [Ald1990]) to generate a random spanning tree with the uniform distribution, as follows.

Start from any vertex. Perform a random walk by choosing at every step one neighbor uniformly at random. Every time a new vertex j is met, add the edge (i, j) to the spanning tree, where i is the previous vertex in the random walk.

INPUT:

- `output_as_graph` – boolean (default: `False`); whether to return a list of edges or a graph

See also:

spanning_trees_count() and *spanning_trees()*

EXAMPLES:

```
sage: G = graphs.TietzeGraph()
sage: G.random_spanning_tree(output_as_graph=True)
Graph on 12 vertices
sage: rg = G.random_spanning_tree(); rg # random
[(0, 9),
(9, 11),
(0, 8),
(8, 7),
```

(continues on next page)

(continued from previous page)

```
(7, 6),
(7, 2),
(2, 1),
(1, 5),
(9, 10),
(5, 4),
(2, 3)]
sage: Graph(rg).is_tree()
True
```

A visual example for the grid graph:

```
sage: G = graphs.Grid2dGraph(6, 6)
sage: pos = G.get_pos()
sage: T = G.random_spanning_tree(True)
sage: T.set_pos(pos)
sage: T.show(vertex_labels=False)
```

rank_decomposition(*G*, *verbose=False*)

Compute an optimal rank-decomposition of the given graph.

This function is available as a method of the *Graph* class. See *rank_decomposition*.

INPUT:

- verbose – boolean (default: False); whether to display progress information while computing the decomposition

OUTPUT:

A pair (rankwidth, decomposition_tree), where rankwidth is a numerical value and decomposition_tree is a ternary tree describing the decomposition (cf. the module's documentation).

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.rankwidth import rank_
↪decomposition
sage: g = graphs.PetersenGraph()
sage: rank_decomposition(g)
(3, Graph on 19 vertices)
```

On more than 32 vertices:

```
sage: g = graphs.RandomGNP(40, .5)
sage: rank_decomposition(g)
Traceback (most recent call last):
...
RuntimeError: the rank decomposition cannot be computed on graphs of >= 32␣
↪vertices
```

The empty graph:

```
sage: g = Graph()
sage: rank_decomposition(g)
(0, Graph on 0 vertices)
```

seidel_adjacency_matrix(*vertices=None*)

Return the Seidel adjacency matrix of self.

Returns $J - I - 2A$, for A the (ordinary) *adjacency matrix* of `self`, I the identity matrix, and J the all-1 matrix. It is closely related to *twograph()*.

The matrix returned is over the integers. If a different ring is desired, use either the `sage.matrix.matrix0.Matrix.change_ring()` method or the `matrix()` function.

INPUT:

- `vertices` – list of vertices (default: `None`); the ordering of the vertices defining how they should appear in the matrix. By default, the ordering given by *vertices()* is used.

EXAMPLES:

```
sage: G = graphs.CycleGraph(5)
sage: G = G.disjoint_union(graphs.CompleteGraph(1))
sage: G.seidel_adjacency_matrix().minpoly()
x^2 - 5
```

seidel_switching(*s*, *inplace=True*)

Return the Seidel switching of `self` w.r.t. subset of vertices `s`.

Returns the graph obtained by Seidel switching of `self` with respect to the subset of vertices `s`. This is the graph given by Seidel adjacency matrix DSD, for S the Seidel adjacency matrix of `self`, and D the diagonal matrix with -1s at positions corresponding to `s`, and 1s elsewhere.

INPUT:

- `s` – a list of vertices of `self`.
- `inplace` – boolean (default: `True`); whether to do the modification inplace, or to return a copy of the graph after switching.

EXAMPLES:

```
sage: G = graphs.CycleGraph(5)
sage: G = G.disjoint_union(graphs.CompleteGraph(1))
sage: G.seidel_switching([(0,1),(1,0),(0,0)])
sage: G.seidel_adjacency_matrix().minpoly()
x^2 - 5
sage: G.is_connected()
True
```

spanning_trees(*labels=False*)

Returns a list of all spanning trees.

If the graph is disconnected, returns the empty list.

Uses the Read-Tarjan backtracking algorithm [RT1975a].

INPUT:

- `labels` – boolean (default: `False`); whether to return edges labels in the spanning trees or not

EXAMPLES:

```
sage: G = Graph([(1,2),(1,2),(1,3),(1,3),(2,3),(1,4)], multiedges=True)
sage: len(G.spanning_trees())
8
sage: G.spanning_trees_count()
8
sage: G = Graph([(1,2),(2,3),(3,1),(3,4),(4,5),(4,5),(4,6)], multiedges=True)
```

(continues on next page)

(continued from previous page)

```
sage: len(G.spanning_trees())
6
sage: G.spanning_trees_count()
6
```

See also:

- *spanning_trees_count()* – counts the number of spanning trees.
- *random_spanning_tree()* – returns a random spanning tree.

sparse6_string()

Return the sparse6 representation of the graph as an ASCII string.

Only valid for undirected graphs on 0 to 262143 vertices, but loops and multiple edges are permitted.

Note: As the sparse6 format only handles graphs whose vertex set is $\{0, ..., n-1\}$, a *relabelled copy* of your graph will be encoded if necessary.

EXAMPLES:

```
sage: G = graphs.BullGraph()
sage: G.sparse6_string()
':Da@en'
```

```
sage: G = Graph(loops=True, multiedges=True, data_structure="sparse")
sage: Graph(':?', data_structure="sparse") == G
True
```

spqr_tree(*G*, *algorithm='Hopcroft_Tarjan'*, *solver=None*, *verbose=0*)

Return an SPQR-tree representing the triconnected components of the graph.

An SPQR-tree is a tree data structure used to represent the triconnected components of a biconnected (multi)graph and the 2-vertex cuts separating them. A node of a SPQR-tree, and the graph associated with it, can be one of the following four types:

- `"S"` – the associated graph is a cycle with at least three vertices. `"S"` stands for `series`.
- `"P"` – the associated graph is a dipole graph, a multigraph with two vertices and three or more edges. `"P"` stands for `parallel`.
- `"Q"` – the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge.
- `"R"` – the associated graph is a 3-connected graph that is not a cycle or dipole. `"R"` stands for `rigid`.

This method decomposes a biconnected graph into cycles, cocycles, and 3-connected blocks summed over cocycles, and arranges them as a SPQR-tree. More precisely, it splits the graph at each of its 2-vertex cuts, giving a unique decomposition into 3-connected blocks, cycles and cocycles. The cocycles are dipole graphs with one edge per real edge between the included vertices and one additional (virtual) edge per connected component resulting from deletion of the vertices in the cut. See the Wikipedia article SPQR_tree.

INPUT:

- `G` – the input graph
- `algorithm` – string (default: `"Hopcroft_Tarjan"`); the algorithm to use among:

  - "Hopcroft_Tarjan" (default) – use the algorithm proposed by Hopcroft and Tarjan in [Hopcroft1973] and later corrected by Gutwenger and Mutzel in [Gut2001]. See *TriconnectivitySPQR*.
  - "cleave" – using method *cleave()*

- solver – string (default: None); specifies a Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method sage.numerical.mip.MixedIntegerLinearProgram.solve() of the class sage.numerical.mip.MixedIntegerLinearProgram.
- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT: SPQR-tree a tree whose vertices are labeled with the block's type and the subgraph of three-blocks in the decomposition.

EXAMPLES:

```
sage: from sage.graphs.connectivity import spqr_tree
sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: Tree = spqr_tree(G)
sage: Tree.order()
4
sage: K4 = graphs.CompleteGraph(4)
sage: all(u[1].is_isomorphic(K4) for u in Tree if u[0] == 'R')
True
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_path([0, G.add_vertex(), G.add_vertex(), 1])
sage: Tree = spqr_tree(G)
sage: Tree.order()
4
sage: C4 = graphs.CycleGraph(4)
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edge_iterator())
sage: Tree = spqr_tree(G)
sage: Tree.order()
13
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = graphs.CycleGraph(6)
sage: Tree = spqr_tree(G)
sage: Tree.order()
1
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
```

(continues on next page)

(continued from previous page)

```
True
sage: G.add_edge(0, 3)
sage: Tree = spqr_tree(G)
sage: Tree.order()
3
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph('LlCG{O@?GBoMw?')
sage: T = spqr_tree(G, algorithm="Hopcroft_Tarjan")
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: T2 = spqr_tree(G, algorithm='cleave')
sage: G.is_isomorphic(spqr_tree_to_graph(T2))
True

sage: G = Graph([(0, 1)], multiedges=True)
sage: T = spqr_tree(G, algorithm='cleave')
sage: T.vertices()
[('Q', Multi-graph on 2 vertices)]
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: T = spqr_tree(G, algorithm='Hopcroft_Tarjan')
sage: T.vertices()
[('Q', Multi-graph on 2 vertices)]
sage: G.add_edge(0, 1)
sage: spqr_tree(G, algorithm='cleave').vertices()
[('P', Multi-graph on 2 vertices)]

sage: from collections import Counter
sage: G = graphs.PetersenGraph()
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: Counter(u[0] for u in T)
Counter({'R': 1})
sage: T = G.spqr_tree(algorithm="cleave")
sage: Counter(u[0] for u in T)
Counter({'R': 1})
sage: for u,v in list(G.edges(labels=False, sort=False)):
....:     G.add_path([u, G.add_vertex(), G.add_vertex(), v])
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 15), ('R', 1), ('S', 15)]
sage: T = G.spqr_tree(algorithm="cleave")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 15), ('R', 1), ('S', 15)]
sage: for u,v in list(G.edges(labels=False, sort=False)):
....:     G.add_path([u, G.add_vertex(), G.add_vertex(), v])
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 60), ('R', 1), ('S', 75)]
sage: T = G.spqr_tree(algorithm="cleave")       # long time
sage: sorted(Counter(u[0] for u in T).items())  # long time
[('P', 60), ('R', 1), ('S', 75)]
```

strong_orientation()

Returns a strongly connected orientation of the current graph.

An orientation of an undirected graph is a digraph obtained by giving an unique direction to each of its

edges. An orientation is said to be strong if there is a directed path between each pair of vertices. See also the Wikipedia article Strongly_connected_component.

If the graph is 2-edge-connected, a strongly connected orientation can be found in linear time. If the given graph is not 2-connected, the orientation returned will ensure that each 2-connected component has a strongly connected orientation.

OUTPUT:

A digraph representing an orientation of the current graph.

Note:

- This method assumes the graph is connected.
- This algorithm works in O(m).

EXAMPLES:

For a 2-regular graph, a strong orientation gives to each vertex an out-degree equal to 1:

```
sage: g = graphs.CycleGraph(5)
sage: g.strong_orientation().out_degree()
[1, 1, 1, 1, 1]
```

The Petersen Graph is 2-edge connected. It then has a strongly connected orientation:

```
sage: g = graphs.PetersenGraph()
sage: o = g.strong_orientation()
sage: len(o.strongly_connected_components())
1
```

The same goes for the CubeGraph in any dimension

```
sage: all(len(graphs.CubeGraph(i).strong_orientation().strongly_connected_
↪components()) == 1 for i in range(2,6))
True
```

A multigraph also has a strong orientation

```
sage: g = Graph([(1,2),(1,2)], multiedges=True)
sage: g.strong_orientation()
Multi-digraph on 2 vertices
```

strong_orientations_iterator(G)

Returns an iterator over all strong orientations of a graph G.

A strong orientation of a graph is an orientation of its edges such that the obtained digraph is strongly connected (i.e. there exist a directed path between each pair of vertices).

ALGORITHM:

It is an adaptation of the algorithm published in [CGMRV16]. It runs in $O(mn)$ amortized time, where m is the number of edges and n is the number of vertices. The amortized time can be improved to $O(m)$ with a more involved method. In this function, first the graph is preprocessed and a spanning tree is generated. Then every orientation of the non-tree edges of the graph can be extended to at least one new strong orientation by orienting properly the edges of the spanning tree (this property is proved in [CGMRV16]). Therefore, this function generates all partial orientations of the non-tree edges and then launches a helper

function corresponding to the generation algorithm described in [CGMRV16]. In order to avoid trivial symmetries, the orientation of an arbitrary edge is fixed before the start of the enumeration process.

INPUT:

- `G` – an undirected graph.

OUTPUT:

- an iterator which will produce all strong orientations of this graph.

Note: Works only for simple graphs (no multiple edges). To avoid symmetries an orientation of an arbitrary edge is fixed.

EXAMPLES:

A cycle has one possible (non-symmetric) strong orientation:

```
sage: g = graphs.CycleGraph(4)
sage: it = g.strong_orientations_iterator()
sage: len(list(it))
1
```

A tree cannot be strongly oriented:

```
sage: g = graphs.RandomTree(100)
sage: len(list(g.strong_orientations_iterator()))
0
```

Neither can be a disconnected graph:

```
sage: g = graphs.CompleteGraph(6)
sage: g.add_vertex(7)
sage: len(list(g.strong_orientations_iterator()))
0
```

to_directed(*data_structure=None*, *sparse=None*)

Return a directed version of the graph.

A single edge becomes two edges, one in each direction.

INPUT:

- `data_structure` – one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of *Graph* or *DiGraph*.
- `sparse` – boolean (default: `None`); `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`.

EXAMPLES:

```
sage: graphs.PetersenGraph().to_directed()
Petersen graph: Digraph on 10 vertices
```

to_undirected()

Since the graph is already undirected, simply returns a copy of itself.

EXAMPLES:

```
sage: graphs.PetersenGraph().to_undirected()
Petersen graph: Graph on 10 vertices
```

topological_minor(*H*, *vertices=False*, *paths=False*, *solver=None*, *verbose=0*)

Return a topological H-minor from `self` if one exists.

We say that a graph G has a topological H-minor (or that it has a graph isomorphic to H as a topological minor), if G contains a subdivision of a graph isomorphic to H (i.e. obtained from H through arbitrary subdivision of its edges) as a subgraph.

For more information, see the Wikipedia article Minor_(graph_theory).

INPUT:

- `H` – The topological minor to find in the current graph.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

The topological H-minor found is returned as a subgraph M of `self`, such that the vertex v of M that represents a vertex $h \in H$ has `h` as a label (see *get_vertex* and *set_vertex*), and such that every edge of M has as a label the edge of H it (partially) represents.

If no topological minor is found, this method returns `False`.

ALGORITHM:

Mixed Integer Linear Programming.

COMPLEXITY:

Theoretically, when H is fixed, testing for the existence of a topological H-minor is polynomial. The known algorithms are highly exponential in H, though.

Note: This function can be expected to be *very* slow, especially where the topological minor does not exist.

(CPLEX seems to be *much* more efficient than GLPK on this kind of problem)

EXAMPLES:

Petersen's graph has a topological K_4-minor:

```
sage: g = graphs.PetersenGraph()
sage: g.topological_minor(graphs.CompleteGraph(4))
Subgraph of (Petersen graph): Graph on ...
```

And a topological $K_{3,3}$-minor:

```
sage: g.topological_minor(graphs.CompleteBipartiteGraph(3,3))
Subgraph of (Petersen graph): Graph on ...
```

And of course, a tree has no topological C_3-minor:

```
sage: g = graphs.RandomGNP(15,.3)
sage: g = g.subgraph(edges = g.min_spanning_tree())
sage: g.topological_minor(graphs.CycleGraph(3))
False
```

treewidth(*k=None*, *certificate=False*, *algorithm=None*)

Computes the tree-width of G (and provides a decomposition)

INPUT:

- `k` – integer (default: `None`); indicates the width to be considered. When `k` is an integer, the method checks that the graph has treewidth $\leq k$. If `k` is `None` (default), the method computes the optimal tree-width.
- `certificate` – boolean (default: `False`); whether to return the tree-decomposition itself.
- `algorithm` – whether to use `"sage"` or `"tdlib"` (requires the installation of the 'tdlib' package). The default behaviour is to use 'tdlib' if it is available, and Sage's own algorithm when it is not.

OUTPUT:

`g.treewidth()` returns the treewidth of `g`. When `k` is specified, it returns `False` when no tree-decomposition of width $\leq k$ exists or `True` otherwise. When `certificate=True`, the tree-decomposition is also returned.

ALGORITHM:

This function virtually explores the graph of all pairs `(vertex_cut,cc)`, where `vertex_cut` is a vertex cut of the graph of cardinality $\leq k+1$, and `connected_component` is a connected component of the graph induced by `G-vertex_cut`.

We deduce that the pair `(vertex_cut,cc)` is feasible with tree-width k if `cc` is empty, or if a vertex `v` from `vertex_cut` can be replaced with a vertex from `cc`, such that the pair `(vertex_cut+v,cc-v)` is feasible.

Note: The implementation would be much faster if `cc`, the argument of the recursive function, was a bitset. It would also be very nice to not copy the graph in order to compute connected components, for this is really a waste of time.

See also:

`path_decomposition()` computes the pathwidth of a graph. See also the *`vertex_separation`* module.

EXAMPLES:

The PetersenGraph has treewidth 4:

```
sage: graphs.PetersenGraph().treewidth()
4
sage: graphs.PetersenGraph().treewidth(certificate=True)
Tree decomposition: Graph on 6 vertices
```

The treewidth of a 2d grid is its smallest side:

```
sage: graphs.Grid2dGraph(2,5).treewidth()
2
sage: graphs.Grid2dGraph(3,5).treewidth()
3
```

tutte_polynomial(*G*, *edge_selector=None*, *cache=None*)

Return the Tutte polynomial of the graph G.

INPUT:

- `edge_selector` (optional; method) this argument allows the user to specify his own heuristic for selecting edges used in the deletion contraction recurrence
- `cache` – (optional; dict) a dictionary to cache the Tutte polynomials generated in the recursive process. One will be created automatically if not provided.

EXAMPLES:

The Tutte polynomial of any tree of order n is x^{n-1}:

```
sage: all(T.tutte_polynomial() == x**9 for T in graphs.trees(10))
True
```

The Tutte polynomial of the Petersen graph is:

```
sage: P = graphs.PetersenGraph()
sage: P.tutte_polynomial()
x^9 + 6*x^8 + 21*x^7 + 56*x^6 + 12*x^5*y + y^6 + 114*x^5 + 70*x^4*y
+ 30*x^3*y^2 + 15*x^2*y^3 + 10*x*y^4 + 9*y^5 + 170*x^4 + 170*x^3*y
+ 105*x^2*y^2 + 65*x*y^3 + 35*y^4 + 180*x^3 + 240*x^2*y + 171*x*y^2
+ 75*y^3 + 120*x^2 + 168*x*y + 84*y^2 + 36*x + 36*y
```

The Tutte polynomial of G evaluated at (1,1) is the number of spanning trees of G:

```
sage: G = graphs.RandomGNP(10,0.6)
sage: G.tutte_polynomial()(1,1) == G.spanning_trees_count()
True
```

Given that $T(x, y)$ is the Tutte polynomial of a graph G with n vertices and c connected components, then $(-1)^{n-c}x^kT(1-x, 0)$ is the chromatic polynomial of G.

```
sage: G = graphs.OctahedralGraph()
sage: T = G.tutte_polynomial()
sage: R = PolynomialRing(ZZ, 'x')
sage: R((-1)^5*x*T(1-x,0)).factor()
(x - 2) * (x - 1) * x * (x^3 - 9*x^2 + 29*x - 32)
sage: G.chromatic_polynomial().factor()
(x - 2) * (x - 1) * x * (x^3 - 9*x^2 + 29*x - 32)
```

two_factor_petersen(*solver=None*, *verbose=0*)

Return a decomposition of the graph into 2-factors.

Petersen's 2-factor decomposition theorem asserts that any $2r$-regular graph G can be decomposed into 2-factors. Equivalently, it means that the edges of any $2r$-regular graphs can be partitionned in r sets $C_1, \ldots, C_r$ such that for all i, the set C_i is a disjoint union of cycles (a 2-regular graph).

As any graph of maximal degree Δ can be completed into a regular graph of degree $2\lceil\frac{\Delta}{2}\rceil$, this result also means that the edges of any graph of degree Δ can be partitionned in $r = 2\lceil\frac{\Delta}{2}\rceil$ sets $C_1, \ldots, C_r$ such that for all i, the set C_i is a graph of maximal degree 2 (a disjoint union of paths and cycles).

INPUT:

- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.

- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

The Complete Graph on 7 vertices is a 6-regular graph, so it can be edge-partitionned into 2-regular graphs:

```
sage: g = graphs.CompleteGraph(7)
sage: classes = g.two_factor_petersen()
sage: for c in classes:
....:     gg = Graph()
....:     gg.add_edges(c)
....:     print(max(gg.degree())<=2)
True
True
True
sage: Set(set(classes[0]) | set(classes[1]) | set(classes[2])).cardinality()
↪== g.size()
True
```

```
sage: g = graphs.CirculantGraph(24, [7, 11])
sage: cl = g.two_factor_petersen()
sage: g.plot(edge_colors={'black':cl[0], 'red':cl[1]})
Graphics object consisting of 73 graphics primitives
```

twograph()

Return the two-graph of self

Returns the two-graph with the triples $T = \{t \in \binom{V}{3} : \left|\binom{t}{2} \cap E\right| \text{ odd}\}$ where V and E are vertices and edges of self, respectively.

EXAMPLES:

```
sage: p=graphs.PetersenGraph()
sage: p.twograph()
Incidence structure with 10 points and 60 blocks
sage: p=graphs.chang_graphs()
sage: T8 = graphs.CompleteGraph(8).line_graph()
sage: C = T8.seidel_switching([(0,1,None),(2,3,None),(4,5,None),(6,7,None)],
↪inplace=False)
sage: T8.twograph() == C.twograph()
True
sage: T8.is_isomorphic(C)
False
```

See also:

- descendant() – computes the descendant graph of the two-graph of self at a vertex
- twograph_descendant() – ditto, but much faster.

vertex_cover(*algorithm='Cliquer'*, *value_only=False*, *reduction_rules=True*, *solver=None*, *verbosity=0*)

Return a minimum vertex cover of self represented by a set of vertices.

A minimum vertex cover of a graph is a set S of vertices such that each edge is incident to at least one element of S, and such that S is of minimum cardinality. For more information, see the Wikipedia article Vertex_cover.

Equivalently, a vertex cover is defined as the complement of an independent set.

As an optimization problem, it can be expressed as follows:

$$\begin{aligned}\text{Minimize : } & \sum_{v \in G} b_v \\ \text{Such that : } & \forall (u,v) \in G.edges(), b_u + b_v \geq 1 \\ & \forall x \in G, b_x \text{ is a binary variable}\end{aligned}$$

INPUT:

- `algorithm` – string (default: `"Cliquer"`). Indicating which algorithm to use. It can be one of those values.
  - `"Cliquer"` will compute a minimum vertex cover using the Cliquer package.
  - `"MILP"` will compute a minimum vertex cover through a mixed integer linear program.
  - `"mcqd"` will use the MCQD solver (http://www.sicmm.org/~konc/maxclique/). Note that the MCQD package must be installed.
- `value_only` – boolean (default: `False`); if set to `True`, only the size of a minimum vertex cover is returned. Otherwise, a minimum vertex cover is returned as a list of vertices.
- `reduction_rules` – (default: `True`); specify if the reductions rules from kernelization must be applied as pre-processing or not. See [ACFLSS04] for more details. Note that depending on the instance, it might be faster to disable reduction rules.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbosity` – non-negative integer (default: `0`); set the level of verbosity you want from the linear program solver. Since the problem of computing a vertex cover is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver. This option is only useful if `algorithm="MILP"`.

EXAMPLES:

On the Pappus graph:

```
sage: g = graphs.PappusGraph()
sage: g.vertex_cover(value_only=True)
9
```

write_to_eps (*filename*, ***options*)

Write a plot of the graph to `filename` in `eps` format.

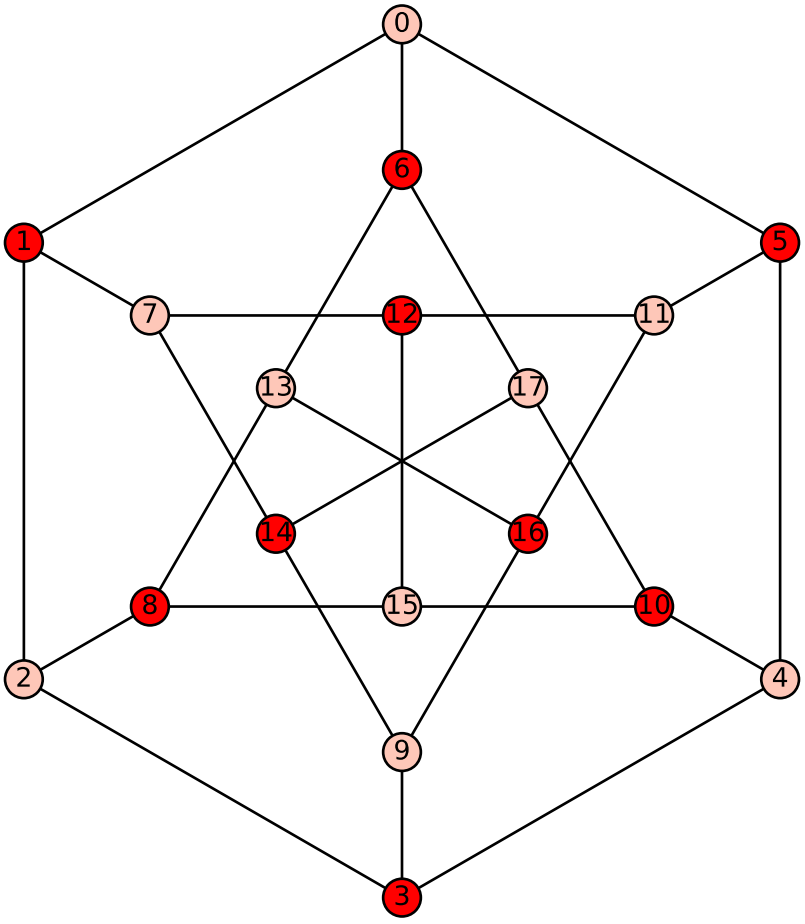
INPUT:

- `filename` – a string
- `**options` – same layout options as *layout()*

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.write_to_eps(tmp_filename(ext='.eps'))
```

It is relatively simple to include this file in a LaTeX document. `\usepackage{graphics}` must appear in the preamble, and `\includegraphics{filename}` will include the file. To compile the document to `pdf` with `pdflatex` or `xelatex` the file needs first to be converted to `pdf`, for example with `ps2pdf filename.eps filename.pdf`.


0
6
1
5
7
12
11
13
17
14
16
8
15
10
2
4
9
3

1.3 Directed graphs

This module implements functions and operations involving directed graphs. Here is what they can do

Graph basic operations:

layout_acyclic_dummy()	Compute a (dummy) ranked layout so that all edges point upward.
layout_acyclic()	Compute a ranked layout so that all edges point upward.
reverse()	Return a copy of digraph with edges reversed in direction.
reverse_edge()	Reverse an edge.
reverse_edges()	Reverse a list of edges.
out_degree_sequence()	Return the outdegree sequence.
out_degree_iterator()	Same as degree_iterator, but for out degree.
out_degree()	Same as degree, but for out degree.
in_degree_sequence()	Return the indegree sequence of this digraph.
in_degree_iterator()	Same as degree_iterator, but for in degree.
in_degree()	Same as degree, but for in-degree.
neighbors_out()	Return the list of the out-neighbors of a given vertex.
neighbor_out_iterator()	Return an iterator over the out-neighbors of a given vertex.
neighbors_in()	Return the list of the in-neighbors of a given vertex.
neighbor_in_iterator()	Return an iterator over the in-neighbors of vertex.
outgoing_edges()	Return a list of edges departing from vertices.
outgoing_edge_iterator()	Return an iterator over all departing edges from vertices
incoming_edges()	Return a list of edges arriving at vertices.
incoming_edge_iterator()	Return an iterator over all arriving edges from vertices
sources()	Return the list of all sources (vertices without incoming edges) of this digraph.
sinks()	Return the list of all sinks (vertices without outgoing edges) of this digraph.
to_undirected()	Return an undirected version of the graph.
to_directed()	Since the graph is already directed, simply returns a copy of itself.
is_directed()	Since digraph is directed, returns True.
dig6_string()	Return the `dig6` representation of the digraph as an ASCII string.

Paths and cycles:

all_paths_iterator()	Return an iterator over the paths of `self`.
all_simple_paths()	Return a list of all the simple paths of `self` starting with one of the given vertices.
all_cycles_iterator()	Return an iterator over all the cycles of `self` starting with one of the given vertices.
all_simple_cycles()	Return a list of all simple cycles of `self`.

Representation theory:

path_semigroup()	Return the (partial) semigroup formed by the paths of the digraph.

Connectivity:

is_strongly_connected()	Check whether the current DiGraph is strongly connected.
strongly_connected_components_digraph()	Return the digraph of the strongly connected components
strongly_connected_components_subgraphs()	Return the strongly connected components as a list of subgraphs.
strongly_connected_component_containing_vertex()	Return the strongly connected component containing a given vertex
strongly_connected_components()	Return the list of strongly connected components.
immediate_dominators()	Return the immediate dominators of all vertices reachable from $root$.
strong_articulation_points()	Return the strong articulation points of this digraph.

Acyclicity:

is_directed_acyclic()	Check whether the digraph is acyclic or not.
is_transitive()	Check whether the digraph is transitive or not.
is_aperiodic()	Check whether the digraph is aperiodic or not.
is_tournament()	Check whether the digraph is a tournament.
period()	Return the period of the digraph.
level_sets()	Return the level set decomposition of the digraph.
topological_sort_generator()	Return a list of all topological sorts of the digraph if it is acyclic
topological_sort()	Return a topological sort of the digraph if it is acyclic

Hard stuff:

feedback_edge_set()	Compute the minimum feedback edge (arc) set of a digraph

Miscellanous:

flow_polytope()	Compute the flow polytope of a digraph
degree_polynomial()	Return the generating polynomial of degrees of vertices in self.
out_branchings()	Return an iterator over the out branchings rooted at given vertex in self.
in_branchings()	Return an iterator over the in branchings rooted at given vertex in self.

1.3.1 Methods

class sage.graphs.digraph.**DiGraph**(*data=None*, *pos=None*, *loops=None*, *format=None*, *weighted=None*, *data_structure='sparse'*, *vertex_labels=True*, *name=None*, *multiedges=None*, *convert_empty_dict_labels_to_None=None*, *sparse=True*, *immutable=False*)

Bases: *sage.graphs.generic_graph.GenericGraph*

Directed graph.

A digraph or directed graph is a set of vertices connected by oriented edges. See also the Wikipedia article Directed_graph. For a collection of pre-defined digraphs, see the *digraph_generators* module.

A *DiGraph* object has many methods whose list can be obtained by typing g.<tab> (i.e. hit the 'tab' key) or by reading the documentation of *digraph*, *generic_graph*, and *graph*.

INPUT:

By default, a *DiGraph* object is simple (i.e. no *loops* nor *multiple edges*) and unweighted. This can be easily tuned with the appropriate flags (see below).

- data – can be any of the following (see the format argument):

1. `DiGraph()` – build a digraph on 0 vertices
2. `DiGraph(5)` – return an edgeless digraph on the 5 vertices 0,...,4
3. `DiGraph([list_of_vertices, list_of_edges])` – return a digraph with given vertices/edges

   To bypass auto-detection, prefer the more explicit `DiGraph([V, E], format='vertices_and_edges')`.
4. `DiGraph(list_of_edges)` – return a digraph with a given list of edges (see documentation of *add_edges()*).

   To bypass auto-detection, prefer the more explicit `DiGraph(L, format='list_of_edges')`.
5. `DiGraph({1: [2,3,4], 3: [4]})` – return a digraph by associating to each vertex the list of its out-neighbors.

   To bypass auto-detection, prefer the more explicit `DiGraph(D, format='dict_of_lists')`.
6. `DiGraph({1: {2: 'a', 3: 'b'}, 3: {2: 'c'}})` – return a digraph by associating a list of out-neighbors to each vertex and providing its edge label.

   To bypass auto-detection, prefer the more explicit `DiGraph(D, format='dict_of_dicts')`.

   For digraphs with multiple edges, you can provide a list of labels instead, e.g.: `DiGraph({1: {2: ['a1', 'a2'], 3:['b']}, 3:{2:['c']}})`.
7. `DiGraph(a_matrix)` – return a digraph with given (weighted) adjacency matrix (see documentation of *adjacency_matrix()*).

   To bypass auto-detection, prefer the more explicit `DiGraph(M, format='adjacency_matrix')`. To take weights into account, use `format='weighted_adjacency_matrix'` instead.
8. `DiGraph(a_nonsquare_matrix)` – return a digraph with given incidence matrix (see documentation of *incidence_matrix()*).

   To bypass auto-detection, prefer the more explicit `DiGraph(M, format='incidence_matrix')`.
9. `DiGraph([V, f])` – return a digraph with a vertex set `V` and an edge u, v whenever $f(u, v)$ is `True`. Example: `DiGraph([ [1..10], lambda x,y: abs(x - y).is_square()])`
10. `DiGraph('FOC@?OC@_?')` – return a digraph from a `dig6` string (see documentation of *dig6_string()*).
11. `DiGraph(another_digraph)` – return a digraph from a Sage (di)graph, pygraphviz digraph, NetworkX digraph, or igraph digraph.

- `pos` – dict (default: `None`); a positioning dictionary. For example, the spring layout from NetworkX for the 5-cycle is:

```
{0: [-0.91679746, 0.88169588],
 1: [ 0.47294849, 1.125     ],
 2: [ 1.125     ,-0.12867615],
 3: [ 0.12743933,-1.125     ],
 4: [-1.125     ,-0.50118505]}
```

- `name` – string (default: `None`); gives the graph a name (e.g., name="complete")
- `loops` – boolean (default: `None`); whether to allow loops (ignored if data is an instance of the DiGraph class)

- `multiedges` – boolean (default: `None`); whether to allow multiple edges (ignored if data is an instance of the DiGraph class)
- `weighted` – boolean (default: `None`); whether digraph thinks of itself as weighted or not. See `self.weighted()`
- `format` – string (default: `None`); if set to `None`, *DiGraph* tries to guess input's format. To avoid this possibly time-consuming step, one of the following values can be specified (see description above): `"int"`, `"dig6"`, `"rule"`, `"list_of_edges"`, `"dict_of_lists"`, `"dict_of_dicts"`, `"adjacency_matrix"`, `"weighted_adjacency_matrix"`, `"incidence_matrix"`, `"NX"`, `"igraph"`.
- `sparse` – boolean (default: `True`); `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`
- `data_structure` – string (default: `"sparse"`); one of the following (for more information, see *overview*):
  - `"dense"` – selects the *dense_graph* backend
  - `"sparse"` – selects the *sparse_graph* backend
  - `"static_sparse"` – selects the *static_sparse_backend* (this backend is faster than the sparse backend and smaller in memory, and it is immutable, so that the resulting graphs can be used as dictionary keys).
- `immutable` – boolean (default: `False`); whether to create a immutable digraph. Note that `immutable=True` is actually a shortcut for `data_structure='static_sparse'`.
- `vertex_labels` – boolean (default: `True`); whether to allow any object as a vertex (slower), or only the integers $0, ..., n-1$, where n is the number of vertices.
- `convert_empty_dict_labels_to_None` – boolean (default: `None`); this arguments sets the default edge labels used by NetworkX (empty dictionaries) to be replaced by `None`, the default Sage edge label. It is set to `True` iff a NetworkX graph is on the input.

EXAMPLES:

1. A dictionary of dictionaries:

```
sage: g = DiGraph({0: {1: 'x', 2: 'z', 3: 'a'}, 2: {5: 'out'}}); g
Digraph on 5 vertices
```

   The labels ('x', 'z', 'a', 'out') are labels for edges. For example, 'out' is the label for the edge from 2 to 5. Labels can be used as weights, if all the labels share some common parent.

2. A dictionary of lists (or iterables):

```
sage: g = DiGraph({0: [1, 2, 3], 2: [4]}); g
Digraph on 5 vertices
sage: g = DiGraph({0: (1, 2, 3), 2: (4,)}); g
Digraph on 5 vertices
```

3. A list of vertices and a function describing adjacencies. Note that the list of vertices and the function must be enclosed in a list (i.e., `[list of vertices, function]`).

   We construct a graph on the integers 1 through 12 such that there is a directed edge from i to j if and only if i divides j:

```
sage: g = DiGraph([[1..12], lambda i,j: i != j and i.divides(j)])
sage: g.vertices()
```

(continues on next page)

(continued from previous page)

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: g.adjacency_matrix()
[0 1 1 1 1 1 1 1 1 1 1 1]
[0 0 0 1 0 1 0 1 0 1 0 1]
[0 0 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
```

4. A Sage matrix: Note: If format is not specified, then Sage assumes a square matrix is an adjacency matrix, and a nonsquare matrix is an incidence matrix.

   - an adjacency matrix:

```
sage: M = Matrix([[0, 1, 1, 1, 0],[0, 0, 0, 0, 0],[0, 0, 0, 0, 1],[0, 0,␣
↪0, 0, 0],[0, 0, 0, 0, 0]]); M
[0 1 1 1 0]
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 0 0]
[0 0 0 0 0]
sage: DiGraph(M)
Digraph on 5 vertices

sage: M = Matrix([[0,1,-1],[-1,0,-1/2],[1,1/2,0]]); M
[   0    1   -1]
[  -1    0 -1/2]
[   1  1/2    0]
sage: G = DiGraph(M,sparse=True,weighted=True); G
Digraph on 3 vertices
sage: G.weighted()
True
```

   - an incidence matrix:

```
sage: M = Matrix(6, [-1,0,0,0,1, 1,-1,0,0,0, 0,1,-1,0,0, 0,0,1,-1,0, 0,0,
↪0,1,-1, 0,0,0,0,0]); M
[-1  0  0  0  1]
[ 1 -1  0  0  0]
[ 0  1 -1  0  0]
[ 0  0  1 -1  0]
[ 0  0  0  1 -1]
[ 0  0  0  0  0]
sage: DiGraph(M)
Digraph on 6 vertices
```

5. A `dig6` string: Sage automatically recognizes whether a string is in `dig6` format, which is a directed version of `graph6`:

```
sage: D = DiGraph('IRAaDCIIOWEOKcPWAo')
sage: D
```

(continues on next page)

(continued from previous page)

```
Digraph on 10 vertices

sage: D = DiGraph('IRAaDCIIOEOKcPWAo')
Traceback (most recent call last):
...
RuntimeError: the string (IRAaDCIIOEOKcPWAo) seems corrupt: for n = 10, the␣
↪string is too short

sage: D = DiGraph("IRAaDCI'OWEOKcPWAo")
Traceback (most recent call last):
...
RuntimeError: the string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

6. A NetworkX XDiGraph:

```
sage: import networkx
sage: g = networkx.MultiDiGraph({0: [1, 2, 3], 2: [4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

7. A NetworkX digraph:

```
sage: import networkx
sage: g = networkx.DiGraph({0: [1, 2, 3], 2: [4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

8. An igraph directed Graph (see also *igraph_graph()*):

```
sage: import igraph                                    # optional - python_
↪igraph
sage: g = igraph.Graph([(0,1),(0,2)], directed=True) # optional - python_
↪igraph
sage: DiGraph(g)                                       # optional - python_
↪igraph
Digraph on 3 vertices
```

If `vertex_labels` is `True`, the names of the vertices are given by the vertex attribute `'name'`, if available:

```
sage: g = igraph.Graph([(0,1),(0,2)], directed=True, vertex_attrs={'name':['a
↪','b','c']})  # optional - python_igraph
sage: DiGraph(g).vertices()
↪              # optional - python_igraph
['a', 'b', 'c']
sage: g = igraph.Graph([(0,1),(0,2)], directed=True, vertex_attrs={'label':['a
↪','b','c']}) # optional - python_igraph
sage: DiGraph(g).vertices()
↪              # optional - python_igraph
[0, 1, 2]
```

If the igraph Graph has edge attributes, they are used as edge labels:

```
sage: g = igraph.Graph([(0,1),(0,2)], directed=True, edge_attrs={'name':['a',
↪'b'], 'weight':[1,3]}) # optional - python_igraph
```

(continues on next page)

(continued from previous page)

```
sage: DiGraph(g).edges()
↪                          # optional - python_igraph
[(0, 1, {'name': 'a', 'weight': 1}), (0, 2, {'name': 'b', 'weight': 3})]
```

all_cycles_iterator(*starting_vertices=None*, *simple=False*, *rooted=False*, *max_length=None*, *trivial=False*)

Return an iterator over all the cycles of `self` starting with one of the given vertices.

The cycles are enumerated in increasing length order.

INPUT:

- `starting_vertices` – iterable (default: `None`); vertices from which the cycles must start. If `None`, then all vertices of the graph can be starting points. This argument is necessary if `rooted` is set to `True`.
- `simple` – boolean (default: `False`); if set to `True`, then only simple cycles are considered. A cycle is simple if the only vertex occuring twice in it is the starting and ending one.
- `rooted` – boolean (default: `False`); if set to False, then cycles differing only by their starting vertex are considered the same (e.g. `['a', 'b', 'c', 'a']` and `['b', 'c', 'a', 'b']`). Otherwise, all cycles are enumerated.
- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.

OUTPUT:

iterator

See also:

- *all_simple_cycles()*

AUTHOR:

Alexandre Blondin Masse

EXAMPLES:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: it = g.all_cycles_iterator()
sage: for _ in range(7): print(next(it))
['a', 'a']
['a', 'a', 'a']
['c', 'd', 'c']
['a', 'a', 'a', 'a']
['a', 'a', 'a', 'a', 'a']
['c', 'd', 'c', 'd', 'c']
['a', 'a', 'a', 'a', 'a', 'a']
```

There are no cycles in the empty graph and in acyclic graphs:

```
sage: g = DiGraph()
sage: it = g.all_cycles_iterator()
sage: list(it)
[]
```

(continues on next page)

(continued from previous page)

```
sage: g = DiGraph({0:[1]})
sage: it = g.all_cycles_iterator()
sage: list(it)
[]
```

It is possible to restrict the starting vertices of the cycles:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: it = g.all_cycles_iterator(starting_vertices=['b', 'c'])
sage: for _ in range(3): print(next(it))
['c', 'd', 'c']
['c', 'd', 'c', 'd', 'c']
['c', 'd', 'c', 'd', 'c', 'd', 'c']
```

Also, one can bound the length of the cycles:

```
sage: it = g.all_cycles_iterator(max_length=3)
sage: list(it)
[['a', 'a'], ['a', 'a', 'a'], ['c', 'd', 'c'],
 ['a', 'a', 'a', 'a']]
```

By default, cycles differing only by their starting point are not all enumerated, but this may be parametrized:

```
sage: it = g.all_cycles_iterator(max_length=3, rooted=False)
sage: list(it)
[['a', 'a'], ['a', 'a', 'a'], ['c', 'd', 'c'],
 ['a', 'a', 'a', 'a']]
sage: it = g.all_cycles_iterator(max_length=3, rooted=True)
sage: list(it)
[['a', 'a'], ['a', 'a', 'a'], ['c', 'd', 'c'], ['d', 'c', 'd'],
 ['a', 'a', 'a', 'a']]
```

One may prefer to enumerate simple cycles, i.e. cycles such that the only vertex occuring twice in it is the starting and ending one (see also *all_simple_cycles()*):

```
sage: it = g.all_cycles_iterator(simple=True)
sage: list(it)
[['a', 'a'], ['c', 'd', 'c']]
sage: g = digraphs.Circuit(4)
sage: list(g.all_cycles_iterator(simple=True))
[[0, 1, 2, 3, 0]]
```

all_paths_iterator(*starting_vertices=None*, *ending_vertices=None*, *simple=False*, *max_length=None*, *trivial=False*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return an iterator over the paths of `self`.

The paths are enumerated in increasing length order.

INPUT:

- `starting_vertices` – iterable (default: `None`); vertices from which the paths must start. If `None`, then all vertices of the graph can be starting points.
- `ending_vertices` – iterable (default: `None`); allowed ending vertices of the paths. If `None`, then all vertices are allowed.

- `simple` – boolean (default: `False`); if set to `True`, then only simple paths are considered. Simple paths are paths in which no two arcs share a head or share a tail, i.e. every vertex in the path is entered at most once and exited at most once.
- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`
  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

OUTPUT:

iterator

AUTHOR:

Alexandre Blondin Masse

EXAMPLES:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['d
↪'], report_edges=True, simple=True)
sage: list(pi)
[[('a', 'b'), ('b', 'c'), ('c', 'd')]]

sage: g = DiGraph([(0, 1, 'a'), (0, 1, 'b'), (1, 2,'c'), (1, 2,'d')],␣
↪multiedges=True)
sage: pi =  g.all_paths_iterator(starting_vertices=[0], use_multiedges=True)
sage: for _ in range(6):
....:     print(next(pi))
[0, 1]
[0, 1]
[0, 1, 2]
[0, 1, 2]
[0, 1, 2]
[0, 1, 2]
sage: pi =  g.all_paths_iterator(starting_vertices=[0], use_multiedges=True,␣
↪report_edges=True, labels=True)
sage: for _ in range(6):
....:     print(next(pi))
[(0, 1, 'b')]
[(0, 1, 'a')]
[(0, 1, 'b'), (1, 2, 'd')]
```

(continues on next page)

(continued from previous page)

```
[(0, 1, 'b'), (1, 2, 'c')]
[(0, 1, 'a'), (1, 2, 'd')]
[(0, 1, 'a'), (1, 2, 'c')]
sage: list(g.all_paths_iterator(starting_vertices=[0, 1], ending_vertices=[2],
↪ use_multiedges=False, report_edges=True, labels=True, simple=True))
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
sage: list(g.all_paths_iterator(starting_vertices=[0, 1], ending_vertices=[2],
↪ use_multiedges=False, report_edges=False, labels=True))
[[1, 2], [0, 1, 2]]
sage: list(g.all_paths_iterator(use_multiedges=True, report_edges=False,␣
↪labels=True, max_length=1))
[[1, 2], [1, 2], [0, 1], [0, 1]]
sage: list(g.all_paths_iterator(use_multiedges=True, report_edges=True,␣
↪labels=True, max_length=1))
[[(1, 2, 'd')], [(1, 2, 'c')], [(0, 1, 'b')], [(0, 1, 'a')]]

sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: pi = g.all_paths_iterator()
sage: for _ in range(7):   # py2
....:     print(next(pi))  # py2
['d', 'c']
['b', 'c']
['c', 'd']
['a', 'a']
['a', 'b']
['a', 'a', 'a']
['a', 'a', 'b']
sage: pi = g.all_paths_iterator()
sage: [len(next(pi)) - 1 for _ in range(7)]
[1, 1, 1, 1, 1, 2, 2]
```

It is possible to precise the allowed starting and/or ending vertices:

```
sage: pi = g.all_paths_iterator(starting_vertices=['a'])
sage: for _ in range(5):   # py2
....:     print(next(pi))  # py2
['a', 'a']
['a', 'b']
['a', 'a', 'a']
['a', 'a', 'b']
['a', 'b', 'c']
sage: pi = g.all_paths_iterator(starting_vertices=['a'])
sage: [len(next(pi)) - 1 for _ in range(5)]
[1, 1, 2, 2, 2]
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b
↪'])
sage: for _ in range(5):
....:     print(next(pi))
['a', 'b']
['a', 'a', 'b']
['a', 'a', 'a', 'b']
['a', 'a', 'a', 'a', 'b']
['a', 'a', 'a', 'a', 'a', 'b']
```

One may prefer to enumerate only simple paths (see *all_simple_paths()*):

```
sage: pi = g.all_paths_iterator(simple=True)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'c'],
    ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'c'], ['d', 'c', 'd'],
    ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True)
sage: [len(p) - 1 for p in pi]
[1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
```

Or simply bound the length of the enumerated paths:

```
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b',
↪ 'c'], max_length=6)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'b'], ['a', 'a', 'b'], ['a', 'b', 'c'], ['a', 'a', 'a', 'b'],
    ['a', 'a', 'b', 'c'], ['a', 'a', 'a', 'a', 'b'],
    ['a', 'a', 'a', 'b', 'c'], ['a', 'b', 'c', 'd', 'c'],
    ['a', 'a', 'a', 'a', 'a', 'b'], ['a', 'a', 'a', 'a', 'b', 'c'],
    ['a', 'a', 'b', 'c', 'd', 'c'],
    ['a', 'a', 'a', 'a', 'a', 'a', 'b'],
    ['a', 'a', 'a', 'a', 'a', 'b', 'c'],
    ['a', 'a', 'a', 'b', 'c', 'd', 'c'],
    ['a', 'b', 'c', 'd', 'c', 'd', 'c']]
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b',
↪ 'c'], max_length=6)
sage: [len(p) - 1 for p in pi]
[1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6]
```

By default, empty paths are not enumerated, but it may be parametrized:

```
sage: pi = g.all_paths_iterator(simple=True, trivial=True)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a'], ['b'], ['c'], ['d'], ['a', 'a'], ['a', 'b'], ['b', 'c'],
    ['c', 'd'], ['d', 'c'], ['a', 'b', 'c'], ['b', 'c', 'd'],
    ['c', 'd', 'c'], ['d', 'c', 'd'], ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True, trivial=True)
sage: [len(p) - 1 for p in pi]
[0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
sage: pi = g.all_paths_iterator(simple=True, trivial=False)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'c'],
    ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'c'], ['d', 'c', 'd'],
    ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True, trivial=False)
sage: [len(p) - 1 for p in pi]
[1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
```

all_simple_cycles(*starting_vertices=None*, *rooted=False*, *max_length=None*, *trivial=False*)

Return a list of all simple cycles of `self`.

INPUT:

- `starting_vertices` – iterable (default: `None`); vertices from which the cycles must start. If `None`, then all vertices of the graph can be starting points. This argument is necessary if `rooted` is set to `True`.
- `rooted` – boolean (default: `False`); if set to False, then cycles differing only by their starting vertex are considered the same (e.g. `['a', 'b', 'c', 'a']` and `['b', 'c', 'a', 'b']`). Otherwise, all cycles are enumerated.

- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.

OUTPUT:

list

Note: Although the number of simple cycles of a finite graph is always finite, computing all its cycles may take a very long time.

EXAMPLES:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: g.all_simple_cycles()
[['a', 'a'], ['c', 'd', 'c']]
```

The directed version of the Petersen graph:

```
sage: g = graphs.PetersenGraph().to_directed()
sage: g.all_simple_cycles(max_length=4)
[[0, 1, 0], [0, 4, 0], [0, 5, 0], [1, 2, 1], [1, 6, 1], [2, 3, 2],
 [2, 7, 2], [3, 8, 3], [3, 4, 3], [4, 9, 4], [5, 8, 5], [5, 7, 5],
 [6, 8, 6], [6, 9, 6], [7, 9, 7]]
sage: g.all_simple_cycles(max_length=6)
[[0, 1, 0], [0, 4, 0], [0, 5, 0], [1, 2, 1], [1, 6, 1], [2, 3, 2],
 [2, 7, 2], [3, 8, 3], [3, 4, 3], [4, 9, 4], [5, 8, 5], [5, 7, 5],
 [6, 8, 6], [6, 9, 6], [7, 9, 7], [0, 1, 2, 3, 4, 0],
 [0, 1, 2, 7, 5, 0], [0, 1, 6, 8, 5, 0], [0, 1, 6, 9, 4, 0],
 [0, 4, 9, 6, 1, 0], [0, 4, 9, 7, 5, 0], [0, 4, 3, 8, 5, 0],
 [0, 4, 3, 2, 1, 0], [0, 5, 8, 3, 4, 0], [0, 5, 8, 6, 1, 0],
 [0, 5, 7, 9, 4, 0], [0, 5, 7, 2, 1, 0], [1, 2, 3, 8, 6, 1],
 [1, 2, 7, 9, 6, 1], [1, 6, 8, 3, 2, 1], [1, 6, 9, 7, 2, 1],
 [2, 3, 8, 5, 7, 2], [2, 3, 4, 9, 7, 2], [2, 7, 9, 4, 3, 2],
 [2, 7, 5, 8, 3, 2], [3, 8, 6, 9, 4, 3], [3, 4, 9, 6, 8, 3],
 [5, 8, 6, 9, 7, 5], [5, 7, 9, 6, 8, 5], [0, 1, 2, 3, 8, 5, 0],
 [0, 1, 2, 7, 9, 4, 0], [0, 1, 6, 8, 3, 4, 0],
 [0, 1, 6, 9, 7, 5, 0], [0, 4, 9, 6, 8, 5, 0],
 [0, 4, 9, 7, 2, 1, 0], [0, 4, 3, 8, 6, 1, 0],
 [0, 4, 3, 2, 7, 5, 0], [0, 5, 8, 3, 2, 1, 0],
 [0, 5, 8, 6, 9, 4, 0], [0, 5, 7, 9, 6, 1, 0],
 [0, 5, 7, 2, 3, 4, 0], [1, 2, 3, 4, 9, 6, 1],
 [1, 2, 7, 5, 8, 6, 1], [1, 6, 8, 5, 7, 2, 1],
 [1, 6, 9, 4, 3, 2, 1], [2, 3, 8, 6, 9, 7, 2],
 [2, 7, 9, 6, 8, 3, 2], [3, 8, 5, 7, 9, 4, 3],
 [3, 4, 9, 7, 5, 8, 3]]
```

The complete graph (without loops) on 4 vertices:

```
sage: g = graphs.CompleteGraph(4).to_directed()
sage: g.all_simple_cycles()
[[0, 1, 0], [0, 2, 0], [0, 3, 0], [1, 2, 1], [1, 3, 1], [2, 3, 2],
 [0, 1, 2, 0], [0, 1, 3, 0], [0, 2, 1, 0], [0, 2, 3, 0],
 [0, 3, 1, 0], [0, 3, 2, 0], [1, 2, 3, 1], [1, 3, 2, 1],
 [0, 1, 2, 3, 0], [0, 1, 3, 2, 0], [0, 2, 1, 3, 0],
 [0, 2, 3, 1, 0], [0, 3, 1, 2, 0], [0, 3, 2, 1, 0]]
```

If the graph contains a large number of cycles, one can bound the length of the cycles, or simply restrict the possible starting vertices of the cycles:

```
sage: g = graphs.CompleteGraph(20).to_directed()
sage: g.all_simple_cycles(max_length=2)
[[0, 1, 0], [0, 2, 0], [0, 3, 0], [0, 4, 0], [0, 5, 0], [0, 6, 0],
 [0, 7, 0], [0, 8, 0], [0, 9, 0], [0, 10, 0], [0, 11, 0],
 [0, 12, 0], [0, 13, 0], [0, 14, 0], [0, 15, 0], [0, 16, 0],
 [0, 17, 0], [0, 18, 0], [0, 19, 0], [1, 2, 1], [1, 3, 1],
 [1, 4, 1], [1, 5, 1], [1, 6, 1], [1, 7, 1], [1, 8, 1], [1, 9, 1],
 [1, 10, 1], [1, 11, 1], [1, 12, 1], [1, 13, 1], [1, 14, 1],
 [1, 15, 1], [1, 16, 1], [1, 17, 1], [1, 18, 1], [1, 19, 1],
 [2, 3, 2], [2, 4, 2], [2, 5, 2], [2, 6, 2], [2, 7, 2], [2, 8, 2],
 [2, 9, 2], [2, 10, 2], [2, 11, 2], [2, 12, 2], [2, 13, 2],
 [2, 14, 2], [2, 15, 2], [2, 16, 2], [2, 17, 2], [2, 18, 2],
 [2, 19, 2], [3, 4, 3], [3, 5, 3], [3, 6, 3], [3, 7, 3], [3, 8, 3],
 [3, 9, 3], [3, 10, 3], [3, 11, 3], [3, 12, 3], [3, 13, 3],
 [3, 14, 3], [3, 15, 3], [3, 16, 3], [3, 17, 3], [3, 18, 3],
 [3, 19, 3], [4, 5, 4], [4, 6, 4], [4, 7, 4], [4, 8, 4], [4, 9, 4],
 [4, 10, 4], [4, 11, 4], [4, 12, 4], [4, 13, 4], [4, 14, 4],
 [4, 15, 4], [4, 16, 4], [4, 17, 4], [4, 18, 4], [4, 19, 4],
 [5, 6, 5], [5, 7, 5], [5, 8, 5], [5, 9, 5], [5, 10, 5],
 [5, 11, 5], [5, 12, 5], [5, 13, 5], [5, 14, 5], [5, 15, 5],
 [5, 16, 5], [5, 17, 5], [5, 18, 5], [5, 19, 5], [6, 7, 6],
 [6, 8, 6], [6, 9, 6], [6, 10, 6], [6, 11, 6], [6, 12, 6],
 [6, 13, 6], [6, 14, 6], [6, 15, 6], [6, 16, 6], [6, 17, 6],
 [6, 18, 6], [6, 19, 6], [7, 8, 7], [7, 9, 7], [7, 10, 7],
 [7, 11, 7], [7, 12, 7], [7, 13, 7], [7, 14, 7], [7, 15, 7],
 [7, 16, 7], [7, 17, 7], [7, 18, 7], [7, 19, 7], [8, 9, 8],
 [8, 10, 8], [8, 11, 8], [8, 12, 8], [8, 13, 8], [8, 14, 8],
 [8, 15, 8], [8, 16, 8], [8, 17, 8], [8, 18, 8], [8, 19, 8],
 [9, 10, 9], [9, 11, 9], [9, 12, 9], [9, 13, 9], [9, 14, 9],
 [9, 15, 9], [9, 16, 9], [9, 17, 9], [9, 18, 9], [9, 19, 9],
 [10, 11, 10], [10, 12, 10], [10, 13, 10], [10, 14, 10],
 [10, 15, 10], [10, 16, 10], [10, 17, 10], [10, 18, 10],
 [10, 19, 10], [11, 12, 11], [11, 13, 11], [11, 14, 11],
 [11, 15, 11], [11, 16, 11], [11, 17, 11], [11, 18, 11],
 [11, 19, 11], [12, 13, 12], [12, 14, 12], [12, 15, 12],
 [12, 16, 12], [12, 17, 12], [12, 18, 12], [12, 19, 12],
 [13, 14, 13], [13, 15, 13], [13, 16, 13], [13, 17, 13],
 [13, 18, 13], [13, 19, 13], [14, 15, 14], [14, 16, 14],
 [14, 17, 14], [14, 18, 14], [14, 19, 14], [15, 16, 15],
 [15, 17, 15], [15, 18, 15], [15, 19, 15], [16, 17, 16],
 [16, 18, 16], [16, 19, 16], [17, 18, 17], [17, 19, 17],
 [18, 19, 18]]
sage: g = graphs.CompleteGraph(20).to_directed()
sage: g.all_simple_cycles(max_length=2, starting_vertices=[0])
[[0, 1, 0], [0, 2, 0], [0, 3, 0], [0, 4, 0], [0, 5, 0], [0, 6, 0],
 [0, 7, 0], [0, 8, 0], [0, 9, 0], [0, 10, 0], [0, 11, 0],
 [0, 12, 0], [0, 13, 0], [0, 14, 0], [0, 15, 0], [0, 16, 0],
 [0, 17, 0], [0, 18, 0], [0, 19, 0]]
```

One may prefer to distinguish equivalent cycles having distinct starting vertices (compare the following examples):

```
sage: g = graphs.CompleteGraph(4).to_directed()
sage: g.all_simple_cycles(max_length=2, rooted=False)
[[0, 1, 0], [0, 2, 0], [0, 3, 0], [1, 2, 1], [1, 3, 1], [2, 3, 2]]
```

(continues on next page)

(continued from previous page)

```
sage: g.all_simple_cycles(max_length=2, rooted=True)
[[0, 1, 0], [0, 2, 0], [0, 3, 0], [1, 0, 1], [1, 2, 1], [1, 3, 1],
 [2, 0, 2], [2, 1, 2], [2, 3, 2], [3, 0, 3], [3, 1, 3], [3, 2, 3]]
```

all_simple_paths(*starting_vertices=None*, *ending_vertices=None*, *max_length=None*, *trivial=False*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return a list of all the simple paths of `self` starting with one of the given vertices.

Simple paths are paths in which no two arcs share a head or share a tail, i.e. every vertex in the path is entered at most once and exited at most once.

INPUT:

- `starting_vertices` – list (default: `None`); vertices from which the paths must start. If `None`, then all vertices of the graph can be starting points.
- `ending_vertices` – iterable (default: `None`); allowed ending vertices of the paths. If `None`, then all vertices are allowed.
- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`
  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

OUTPUT:

list

Note: Although the number of simple paths of a finite graph is always finite, computing all its paths may take a very long time.

EXAMPLES:

```
sage: g = DiGraph({0: [0, 1], 1: [2], 2: [3], 3: [2]}, loops=True)
sage: g.all_simple_paths()
[[3, 2],
 [2, 3],
 [1, 2],
 [0, 0],
 [0, 1],
 [0, 1, 2],
 [1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
 [2, 3, 2],
 [3, 2, 3],
 [0, 1, 2, 3]]

sage: g = DiGraph([(0, 1, 'a'), (0, 1, 'b'), (1, 2,'c'), (1, 2,'d')],␣
↪multiedges=True)
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=False)
[[0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True, report_edges=True)
[[(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True, report_edges=True, labels=True)
[[(0, 1, 'b'), (1, 2, 'd')],
 [(0, 1, 'b'), (1, 2, 'c')],
 [(0, 1, 'a'), (1, 2, 'd')],
 [(0, 1, 'a'), (1, 2, 'c')]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=True, labels=True)
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=False, labels=True)
[[1, 2], [0, 1, 2]]
sage: g.all_simple_paths(use_multiedges=True, report_edges=False, labels=True)
[[1, 2], [1, 2], [0, 1], [0, 1], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=True, labels=True, trivial=True)
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
```

One may compute all paths having specific starting and/or ending vertices:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: g.all_simple_paths(starting_vertices=['a'])
[['a', 'a'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'd']]
sage: g.all_simple_paths(starting_vertices=['a'], ending_vertices=['c'])
[['a', 'b', 'c']]
sage: g.all_simple_paths(starting_vertices=['a'], ending_vertices=['b', 'c'])
[['a', 'b'], ['a', 'b', 'c']]
```

It is also possible to bound the length of the paths:

```
sage: g = DiGraph({0: [0, 1], 1: [2], 2: [3], 3: [2]}, loops=True)
sage: g.all_simple_paths(max_length=2)
[[3, 2],
 [2, 3],
 [1, 2],
 [0, 0],
 [0, 1],
 [0, 1, 2],
 [1, 2, 3],
 [2, 3, 2],
 [3, 2, 3]]
```

By default, empty paths are not enumerated, but this can be parametrized:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: g.all_simple_paths(starting_vertices=['a'], trivial=True)
[['a'], ['a', 'a'], ['a', 'b'], ['a', 'b', 'c'],
 ['a', 'b', 'c', 'd']]
sage: g.all_simple_paths(starting_vertices=['a'], trivial=False)
[['a', 'a'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'd']]
```

degree_polynomial()

Return the generating polynomial of degrees of vertices in `self`.

This is the sum

$$\sum_{v \in G} x^{\mathrm{in}(v)} y^{\mathrm{out}(v)},$$

where `in(v)` and `out(v)` are the number of incoming and outgoing edges at vertex v in the digraph G.

Because this polynomial is multiplicative for Cartesian product of digraphs, it is useful to help see if the digraph can be isomorphic to a Cartesian product.

See also:

`num_verts()` for the value at $(x, y) = (1, 1)$

EXAMPLES:

```
sage: G = posets.PentagonPoset().hasse_diagram()
sage: G.degree_polynomial()
x^2 + 3*x*y + y^2

sage: G = posets.BooleanLattice(4).hasse_diagram()
sage: G.degree_polynomial().factor()
(x + y)^4
```

dig6_string()

Return the `dig6` representation of the digraph as an ASCII string.

This is only valid for single (no multiple edges) digraphs on at most $2^{18} - 1 = 262143$ vertices.

Note: As the `dig6` format only handles graphs with vertex set $\{0, \ldots, n-1\}$, a *relabelled copy* will be encoded, if necessary.

See also:

- *graph6_string()* – a similar string format for undirected graphs

EXAMPLES:

```
sage: D = DiGraph({0: [1, 2], 1: [2], 2: [3], 3: [0]})
sage: D.dig6_string()
'CW`_'
```

feedback_edge_set(*constraint_generation=True*, *value_only=False*, *solver=None*, *verbose=0*)

Compute the minimum feedback edge set of a digraph (also called feedback arc set).

The minimum feedback edge set of a digraph is a set of edges that intersect all the circuits of the digraph. Equivalently, a minimum feedback arc set of a DiGraph is a set S of arcs such that the digraph $G - S$ is acyclic. For more information, see the Wikipedia article Feedback_arc_set.

INPUT:

- `value_only` – boolean (default: `False`)
  - When set to `True`, only the minimum cardinal of a minimum edge set is returned.
  - When set to `False`, the `Set` of edges of a minimal edge set is returned.
- `constraint_generation` – boolean (default: `True`); whether to use constraint generation when solving the Mixed Integer Linear Program.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

ALGORITHM:

This problem is solved using Linear Programming, in two different ways. The first one is to solve the following formulation:

$$\begin{aligned}\text{Minimize : } & \sum_{(u,v)\in G} b_{(u,v)} \\ \text{Such that : } & \\ & \forall (u,v) \in G, d_u - d_v + n \cdot b_{(u,v)} \geq 0 \\ & \forall u \in G, 0 \leq d_u \leq |G|\end{aligned}$$

An explanation:

An acyclic digraph can be seen as a poset, and every poset has a linear extension. This means that in any acyclic digraph the vertices can be ordered with a total order $<$ in such a way that if $(u, v) \in G$, then $u < v$.

Thus, this linear program is built in order to assign to each vertex v a number $d_v \in [0, \ldots, n-1]$ such that if there exists an edge $(u, v) \in G$ such that $d_v < d_u$, then the edge (u, v) is removed.

The number of edges removed is then minimized, which is the objective.

(Constraint Generation)

If the parameter `constraint_generation` is enabled, a more efficient formulation is used :

$$\begin{aligned}\text{Minimize : } & \sum_{(u,v)\in G} b_{(u,v)} \\ \text{Such that : } & \\ & \forall C \text{ circuits } \subseteq G, \sum_{uv\in C} b_{(u,v)} \geq 1\end{aligned}$$

As the number of circuits contained in a graph is exponential, this LP is solved through constraint generation. This means that the solver is sequentially asked to solved the problem, knowing only a portion of the circuits contained in G, each time adding to the list of its constraints the circuit which its last answer had left intact.

EXAMPLES:

If the digraph is created from a graph, and hence is symmetric (if uv is an edge, then vu is an edge too), then obviously the cardinality of its feedback arc set is the number of edges in the first graph:

```
sage: cycle=graphs.CycleGraph(5)
sage: dcycle=DiGraph(cycle)
sage: cycle.size()
5
sage: dcycle.feedback_edge_set(value_only=True)
5
```

And in this situation, for any edge uv of the first graph, uv of vu is in the returned feedback arc set:

```
sage: g = graphs.RandomGNP(5,.3)
sage: dg = DiGraph(g)
sage: feedback = dg.feedback_edge_set()
sage: u,v,l = next(g.edge_iterator())
sage: (u,v) in feedback or (v,u) in feedback
True
```

flow_polytope(*edges=None*, *ends=None*, *backend=None*)

Return the flow polytope of a digraph.

The flow polytope of a directed graph is the polytope consisting of all nonnegative flows on the graph with a given set S of sources and a given set T of sinks.

A *flow* on a directed graph G with a given set S of sources and a given set T of sinks means an assignment of a nonnegative real to each edge of G such that the flow is conserved in each vertex outside of S and T, and there is a unit of flow entering each vertex in S and a unit of flow leaving each vertex in T. These flows clearly form a polytope in the space of all assignments of reals to the edges of G.

The polytope is empty unless the sets S and T are equinumerous.

By default, S is taken to be the set of all sources (i.e., vertices of indegree 0) of G, and T is taken to be the set of all sinks (i.e., vertices of outdegree 0) of G. If a different choice of S and T is desired, it can be specified using the optional `ends` parameter.

The polytope is returned as a polytope in $\mathbf{R}^m$, where m is the number of edges of the digraph `self`. The k-th coordinate of a point in the polytope is the real assigned to the k-th edge of `self`. The order of the edges is the one returned by `self.edges()`. If a different order is desired, it can be specified using the optional `edges` parameter.

The faces and volume of these polytopes are of interest. Examples of these polytopes are the Chan-Robbins-Yuen polytope and the Pitman-Stanley polytope [PS2002].

INPUT:

- `edges` – list (default: `None`); a list of edges of `self`. If not specified, the list of all edges of `self` is used with the default ordering of `self.edges()`. This determines which coordinate of a point in the polytope will correspond to which edge of `self`. It is also possible to specify a list which contains not all edges of `self`; this results in a polytope corresponding to the flows which are 0 on all remaining edges. Notice that the edges entered here must be in the precisely same format as outputted by `self.edges()`; so, if `self.edges()` outputs an edge in the form `(1, 3, None)`, then `(1, 3)` will not do!
- `ends` – (optional, default: `(self.sources(), self.sinks())`) a pair (S, T) of an iterable S and an iterable T.
- `backend` – string or `None` (default); the backend to use; see `sage.geometry.polyhedron.constructor.Polyhedron()`

Note: Flow polytopes can also be built through the `polytopes.<tab>` object:

```
sage: polytopes.flow_polytope(digraphs.Path(5))
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
```

EXAMPLES:

A commutative square:

```
sage: G = DiGraph({1: [2, 3], 2: [4], 3: [4]})
sage: fl = G.flow_polytope(); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
sage: fl.vertices()
(A vertex at (0, 1, 0, 1), A vertex at (1, 0, 1, 0))
```

Using a different order for the edges of the graph:

```
sage: fl = G.flow_polytope(edges=G.edges(key=lambda x: x[0] - x[1])); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
sage: fl.vertices()
(A vertex at (0, 1, 1, 0), A vertex at (1, 0, 0, 1))
```

A tournament on 4 vertices:

```
sage: H = digraphs.TransitiveTournament(4)
sage: fl = H.flow_polytope(); fl
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 4 vertices
sage: fl.vertices()
(A vertex at (0, 0, 1, 0, 0, 0),
 A vertex at (0, 1, 0, 0, 0, 1),
 A vertex at (1, 0, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 1, 0, 1))
```

Restricting to a subset of the edges:

```
sage: fl = H.flow_polytope(edges=[(0, 1, None), (1, 2, None),
....:                             (2, 3, None), (0, 3, None)])
sage: fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1), A vertex at (1, 1, 1, 0))
```

Using a different choice of sources and sinks:

```
sage: fl = H.flow_polytope(ends=([1], [3])); fl
A 1-dimensional polyhedron in QQ^6 defined as the convex hull
of 2 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1, 0, 1), A vertex at (0, 0, 0, 0, 1, 0))
sage: fl = H.flow_polytope(ends=([0, 1], [3])); fl
The empty polyhedron in QQ^6
sage: fl = H.flow_polytope(ends=([3], [0])); fl
The empty polyhedron in QQ^6
sage: fl = H.flow_polytope(ends=([0, 1], [2, 3])); fl
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
```

(continues on next page)

(continued from previous page)

```
of 5 vertices
sage: fl.vertices()
(A vertex at (0, 0, 1, 1, 0, 0),
 A vertex at (0, 1, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 2, 0, 1),
 A vertex at (1, 0, 0, 1, 1, 0),
 A vertex at (0, 1, 0, 1, 0, 1))
sage: fl = H.flow_polytope(edges=[(0, 1, None), (1, 2, None),
....:                             (2, 3, None), (0, 2, None),
....:                             (1, 3, None)],
....:                      ends=([0, 1], [2, 3])); fl
A 2-dimensional polyhedron in QQ^5 defined as the convex hull
of 4 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1, 1),
 A vertex at (1, 2, 1, 0, 0),
 A vertex at (1, 1, 0, 0, 1),
 A vertex at (0, 1, 1, 1, 0))
```

A digraph with one source and two sinks:

```
sage: Y = DiGraph({1: [2], 2: [3, 4]})
sage: Y.flow_polytope()
The empty polyhedron in QQ^3
```

A digraph with one vertex and no edge:

```
sage: Z = DiGraph({1: []})
sage: Z.flow_polytope()
A 0-dimensional polyhedron in QQ^0 defined as the convex hull
of 1 vertex
```

A digraph with multiple edges (trac ticket #28837):

```
sage: G = DiGraph([(0, 1), (0,1)], multiedges=True)
sage: G
Multi-digraph on 2 vertices
sage: P = G.flow_polytope()
sage: P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
sage: P.vertices()
(A vertex at (1, 0), A vertex at (0, 1))
sage: P.lines()
()
```

in_branchings(*source*, *spanning=True*)

Return an iterator over the in branchings rooted at given vertex in `self`.

An in-branching is a directed tree rooted at `source` whose arcs are directed to source from leaves. An in-branching is spanning if it contains all vertices of the digraph.

If no spanning in branching rooted at `source` exist, raises ValueError or return non spanning in branching rooted at `source`, depending on the value of `spanning`.

INPUT:

- `source` – vertex used as the source for all in branchings.

- `spanning` – boolean (default: `True`); if `False` return maximum in branching to `source`. Otherwise, return spanning in branching if exists.

OUTPUT:

An iterator over the in branchings rooted in the given source.

See also:

- *`out_branchings()`* – iterator over out-branchings rooted at given vertex.
- *`spanning_trees()`* – returns all spanning trees.
- *`spanning_trees_count()`* – counts the number of spanning trees.

ALGORITHM:

Recursively computes all in branchings.

At each step:

0. clean the graph (see below)
1. pick an edge e incoming to source
2. find all in branchings that do not contain e by first removing it
3. find all in branchings that do contain e by first merging the end vertices of e

Cleaning the graph implies to remove loops and replace multiedges by a single one with an appropriate label since these lead to similar steps of computation.

EXAMPLES:

A bidirectional 4-cycle:

```
sage: G = DiGraph({1:[2,3], 2:[1,4], 3:[1,4], 4:[2,3]}, format='dict_of_lists
↪')
sage: list(G.in_branchings(1))
[Digraph on 4 vertices,
 Digraph on 4 vertices,
 Digraph on 4 vertices,
 Digraph on 4 vertices]
```

With the Petersen graph turned into a symmetric directed graph:

```
sage: G = graphs.PetersenGraph().to_directed()
sage: len(list(G.in_branchings(0)))
2000
```

With a non connected `DiGraph` and `spanning = True`:

```
sage: G = graphs.PetersenGraph().to_directed() + graphs.PetersenGraph().to_
↪directed()
sage: G.in_branchings(0)
Traceback (most recent call last):
...
ValueError: no spanning in branching to vertex (0) exist
```

With a non connected `DiGraph` and `spanning = False`:

```
sage: g=DiGraph([(1,0), (1,0), (2,1), (3,4)],multiedges=True)
sage: list(g.in_branchings(0,spanning=False))
[Digraph on 3 vertices, Digraph on 3 vertices]
```

With multiedges:

```
sage: G = DiGraph({0:[1,1,1], 1:[2,2]}, format='dict_of_lists',␣
↪multiedges=True)
sage: len(list(G.in_branchings(2)))
6
```

With a DiGraph already being a spanning in branching:

```
sage: G = DiGraph({0:[], 1:[0], 2:[0], 3:[1], 4:[1], 5:[2]}, format='dict_of_
↪lists')
sage: next(G.in_branchings(0)) == G
True
```

in_degree(*vertices=None*, *labels=False*)

Same as degree, but for in degree.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.in_degree(vertices=[0, 1, 2], labels=True)
{0: 2, 1: 2, 2: 2}
sage: D.in_degree()
[2, 2, 2, 2, 1, 1]
sage: G = graphs.PetersenGraph().to_directed()
sage: G.in_degree(0)
3
```

in_degree_iterator(*vertices=None*, *labels=False*)

Same as degree_iterator, but for in degree.

EXAMPLES:

```
sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: sorted(D.in_degree_iterator())
[2, 2, 2, 2, 3, 3, 3, 3]
sage: sorted(D.in_degree_iterator(labels=True))
[((0, 0), 2),
 ((0, 1), 3),
 ((0, 2), 3),
 ((0, 3), 2),
 ((1, 0), 2),
 ((1, 1), 3),
 ((1, 2), 3),
 ((1, 3), 2)]
```

in_degree_sequence()

Return the in-degree sequence.

EXAMPLES:

The in-degree sequences of two digraphs:

```
sage: g = DiGraph({1: [2, 5, 6], 2: [3, 6], 3: [4, 6], 4: [6], 5: [4, 6]})
sage: g.in_degree_sequence()
[5, 2, 1, 1, 1, 0]
```

```
sage: V = [2, 3, 5, 7, 8, 9, 10, 11]
sage: E = [[], [8, 10], [11], [8, 11], [9], [], [], [2, 9, 10]]
sage: g = DiGraph(dict(zip(V, E)))
sage: g.in_degree_sequence()
[2, 2, 2, 2, 1, 0, 0, 0]
```

incoming_edge_iterator(*vertices*, *labels=True*)

Return an iterator over all arriving edges from vertices.

INPUT:

- vertices – a vertex or a list of vertices
- labels – boolean (default: True); whether to return edges as pairs of vertices, or as triples containing the labels

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: for a in D.incoming_edge_iterator([0]):
....:     print(a)
(1, 0, None)
(4, 0, None)
```

incoming_edges(*vertices*, *labels=True*)

Return a list of edges arriving at vertices.

INPUT:

- vertices – a vertex or a list of vertices
- labels – boolean (default: True); whether to return edges as pairs of vertices, or as triples containing the labels.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.incoming_edges([0])
[(1, 0, None), (4, 0, None)]
```

is_aperiodic()

Return whether the current DiGraph is aperiodic.

A directed graph is aperiodic if there is no integer $k > 1$ that divides the length of every cycle in the graph. See the Wikipedia article Aperiodic_graph for more information.

EXAMPLES:

The following graph has period 2, so it is not aperiodic:

```
sage: g = DiGraph({0: [1], 1: [0]})
sage: g.is_aperiodic()
False
```

The following graph has a cycle of length 2 and a cycle of length 3, so it is aperiodic:

```
sage: g = DiGraph({0: [1, 4], 1: [2], 2: [0], 4: [0]})
sage: g.is_aperiodic()
True
```

See also:

period()

is_directed()

Since digraph is directed, return `True`.

EXAMPLES:

```
sage: DiGraph().is_directed()
True
```

is_directed_acyclic(*certificate=False*)

Return whether the digraph is acyclic or not.

A directed graph is acyclic if for any vertex v, there is no directed path that starts and ends at v. Every directed acyclic graph (DAG) corresponds to a partial ordering of its vertices, however multiple dags may lead to the same partial ordering.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

- When `certificate=False`, returns a boolean value.
- When `certificate=True`:
  - If the graph is acyclic, returns a pair `(True, ordering)` where `ordering` is a list of the vertices such that `u` appears before `v` in `ordering` if `u, v` is an edge.
  - Else, returns a pair `(False, cycle)` where `cycle` is a list of vertices representing a circuit in the graph.

EXAMPLES:

At first, the following graph is acyclic:

```
sage: D = DiGraph({0:[1, 2, 3], 4:[2, 5], 1:[8], 2:[7], 3:[7], 5:[6,7], 7:[8],
↪ 6:[9], 8:[10], 9:[10]})
sage: D.plot(layout='circular').show()
sage: D.is_directed_acyclic()
True
```

Adding an edge from 9 to 7 does not change it:

```
sage: D.add_edge(9, 7)
sage: D.is_directed_acyclic()
True
```

We can obtain as a proof an ordering of the vertices such that u appears before v if uv is an edge of the graph:

```
sage: D.is_directed_acyclic(certificate=True)
(True, [4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10])
```

Adding an edge from 7 to 4, though, makes a difference:

```
sage: D.add_edge(7, 4)
sage: D.is_directed_acyclic()
False
```

Indeed, it creates a circuit $7, 4, 5$:

```
sage: D.is_directed_acyclic(certificate=True)
(False, [7, 4, 5])
```

Checking acyclic graphs are indeed acyclic

```
sage: def random_acyclic(n, p):
....:  g = graphs.RandomGNP(n, p)
....:  h = DiGraph()
....:  h.add_edges(((u, v) if u < v else (v, u)) for u, v in g.edge_
↪iterator(labels=False))
....:  return h
...
sage: all(random_acyclic(100, .2).is_directed_acyclic()     # long time
....:      for i in range(50))                               # long time
True
```

is_strongly_connected(*G*)

Check whether the current `DiGraph` is strongly connected.

EXAMPLES:

The circuit is obviously strongly connected:

```
sage: from sage.graphs.connectivity import is_strongly_connected
sage: g = digraphs.Circuit(5)
sage: is_strongly_connected(g)
True
sage: g.is_strongly_connected()
True
```

But a transitive triangle is not:

```
sage: g = DiGraph({0: [1, 2], 1: [2]})
sage: is_strongly_connected(g)
False
```

is_tournament()

Check whether the digraph is a tournament.

A tournament is a digraph in which each pair of distinct vertices is connected by a single arc.

EXAMPLES:

```
sage: g = digraphs.RandomTournament(6)
sage: g.is_tournament()
True
sage: u,v = next(g.edge_iterator(labels=False))
sage: g.add_edge(v, u)
sage: g.is_tournament()
False
sage: g.add_edges([(u, v), (v, u)])
sage: g.is_tournament()
False
```

See also:

- Wikipedia article Tournament_(graph_theory)
- *RandomTournament()*
- *TransitiveTournament()*

is_transitive(*g*, *certificate=False*)

Tests whether the digraph is transitive.

A digraph is transitive if for any pair of vertices $u, v \in G$ linked by a uv-path the edge uv belongs to G.

INPUT:

- `certificate` – whether to return a certificate for negative answers.
  - If `certificate = False` (default), this method returns `True` or `False` according to the graph.
  - If `certificate = True`, this method either returns `True` answers or yield a pair of vertices uv such that there exists a uv-path in G but $uv \notin G$.

EXAMPLES:

```
sage: digraphs.Circuit(4).is_transitive()
False
sage: digraphs.Circuit(4).is_transitive(certificate=True)
(0, 2)
sage: digraphs.RandomDirectedGNP(30,.2).is_transitive()
False
sage: D = digraphs.DeBruijn(5, 2)
sage: D.is_transitive()
False
sage: cert = D.is_transitive(certificate=True)
sage: D.has_edge(*cert)
False
sage: D.shortest_path(*cert) != []
True
sage: digraphs.RandomDirectedGNP(20,.2).transitive_closure().is_transitive()
True
```

layout_acyclic(*rankdir='up'*, ***options*)

Return a ranked layout so that all edges point upward.

To this end, the heights of the vertices are set according to the level set decomposition of the graph (see *level_sets()*).

This is achieved by calling `graphviz` and `dot2tex` if available (see *layout_graphviz()*), and using a spring layout with fixed vertical placement of the vertices otherwise (see *layout_acyclic_dummy()* and *layout_ranked()*).

Non acyclic graphs are partially supported by `graphviz`, which then chooses some edges to point down.

INPUT:

- `rankdir` – string (default: `'up'`); indicates which direction the edges should point toward among `'up'`, `'down'`, `'left'`, or `'right'`
- `**options` – passed down to *layout_ranked()* or *layout_graphviz()*

EXAMPLES:

```
sage: H = DiGraph({0: [1, 2], 1: [3], 2: [3], 3: [], 5: [1, 6], 6: [2, 3]})
```

The actual layout computed depends on whether dot2tex and graphviz are installed, so we don't test its relative values:

```
sage: H.layout_acyclic()
{0: [..., ...], 1: [..., ...], 2: [..., ...], 3: [..., ...], 5: [..., ...],␣
↪6: [..., ...]}

sage: H = DiGraph({0: [1]})
sage: pos = H.layout_acyclic(rankdir='up')
sage: pos[1][1] > pos[0][1] + .5
True
sage: pos = H.layout_acyclic(rankdir='down')
sage: pos[1][1] < pos[0][1] - .5
True
sage: pos = H.layout_acyclic(rankdir='right')
sage: pos[1][0] > pos[0][0] + .5
True
sage: pos = H.layout_acyclic(rankdir='left')
sage: pos[1][0] < pos[0][0] - .5
True
```

layout_acyclic_dummy(*heights=None*, *rankdir='up'*, ***options*)

Return a ranked layout so that all edges point upward.

To this end, the heights of the vertices are set according to the level set decomposition of the graph (see *level_sets()*). This is achieved by a spring layout with fixed vertical placement of the vertices otherwise (see *layout_acyclic_dummy()* and *layout_ranked()*).

INPUT:

- `rankdir` – string (default: `'up'`); indicates which direction the edges should point toward among `'up'`, `'down'`, `'left'`, or `'right'`
- `**options` – passed down to *layout_ranked()*

EXAMPLES:

```
sage: H = DiGraph({0: [1, 2], 1: [3], 2: [3], 3: [], 5: [1, 6], 6: [2, 3]})
sage: H.layout_acyclic_dummy()
{0: [1.00..., 0], 1: [1.00..., 1], 2: [1.51..., 2], 3: [1.50..., 3], 5: [2.01.
↪.., 0], 6: [2.00..., 1]}

sage: H = DiGraph({0: [1]})
sage: H.layout_acyclic_dummy(rankdir='up')
{0: [0.5..., 0], 1: [0.5..., 1]}
sage: H.layout_acyclic_dummy(rankdir='down')
{0: [0.5..., 1], 1: [0.5..., 0]}
sage: H.layout_acyclic_dummy(rankdir='left')
{0: [1, 0.5...], 1: [0, 0.5...]}
sage: H.layout_acyclic_dummy(rankdir='right')
{0: [0, 0.5...], 1: [1, 0.5...]}
sage: H = DiGraph({0: [1, 2], 1: [3], 2: [3], 3: [1], 5: [1, 6], 6: [2, 3]})
sage: H.layout_acyclic_dummy()
Traceback (most recent call last):
...
ValueError: `self` should be an acyclic graph
```

level_sets()

Return the level set decomposition of the digraph.

OUTPUT:

- a list of non empty lists of vertices of this graph

The level set decomposition of the digraph is a list l such that the level $l[i]$ contains all the vertices having all their predecessors in the levels $l[j]$ for $j < i$, and at least one in level $l[i-1]$ (unless $i = 0$).

The level decomposition contains exactly the vertices not occuring in any cycle of the graph. In particular, the graph is acyclic if and only if the decomposition forms a set partition of its vertices, and we recover the usual level set decomposition of the corresponding poset.

EXAMPLES:

```
sage: H = DiGraph({0: [1, 2], 1: [3], 2: [3], 3: [], 5: [1, 6], 6: [2, 3]})
sage: H.level_sets()
[[0, 5], [1, 6], [2], [3]]

sage: H = DiGraph({0: [1, 2], 1: [3], 2: [3], 3: [1], 5: [1, 6], 6: [2, 3]})
sage: H.level_sets()
[[0, 5], [6], [2]]
```

This routine is mostly used for Hasse diagrams of posets:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [3], 3: []})
sage: [len(x) for x in H.level_sets()]
[1, 2, 1]
```

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [4]})
sage: [len(x) for x in H.level_sets()]
[1, 2, 1, 1]
```

Complexity: $O(n+m)$ in time and $O(n)$ in memory (besides the storage of the graph itself), where n and m are respectively the number of vertices and edges (assuming that appending to a list is constant time, which it is not quite).

neighbor_in_iterator(*vertex*)

Return an iterator over the in-neighbors of `vertex`.

An vertex u is an in-neighbor of a vertex v if uv in an edge.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: for a in D.neighbor_in_iterator(0):
....:     print(a)
1
4
```

neighbor_out_iterator(*vertex*)

Return an iterator over the out-neighbors of a given vertex.

A vertex u is an out-neighbor of a vertex v if vu in an edge.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: for a in D.neighbor_out_iterator(0):
....:     print(a)
1
2
3
```

neighbors_in(*vertex*)

Return the list of the in-neighbors of a given vertex.

A vertex u is an in-neighbor of a vertex v if uv in an edge.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.neighbors_in(0)
[1, 4]
```

neighbors_out(*vertex*)

Return the list of the out-neighbors of a given vertex.

A vertex u is an out-neighbor of a vertex v if vu in an edge.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.neighbors_out(0)
[1, 2, 3]
```

out_branchings(*source*, *spanning=True*)

Return an iterator over the out branchings rooted at given vertex in `self`.

An out-branching is a directed tree rooted at `source` whose arcs are directed from source to leaves. An out-branching is spanning if it contains all vertices of the digraph.

If no spanning out branching rooted at `source` exist, raises ValueError or return non spanning out branching rooted at `source`, depending on the value of `spanning`.

INPUT:

- `source` – vertex used as the source for all out branchings.
- `spanning` – boolean (default: `True`); if `False` return maximum out branching from `source`. Otherwise, return spanning out branching if exists.

OUTPUT:

An iterator over the out branchings rooted in the given source.

See also:

- *in_branchings()* – iterator over in-branchings rooted at given vertex.
- *spanning_trees()* – returns all spanning trees.
- *spanning_trees_count()* – counts the number of spanning trees.

ALGORITHM:

Recursively computes all out branchings.

At each step:

0. clean the graph (see below)
1. pick an edge e out of source
2. find all out branchings that do not contain e by first removing it
3. find all out branchings that do contain e by first merging the end vertices of e

Cleaning the graph implies to remove loops and replace multiedges by a single one with an appropriate label since these lead to similar steps of computation.

EXAMPLES:

A bidirectional 4-cycle:

```
sage: G = DiGraph({1:[2,3], 2:[1,4], 3:[1,4], 4:[2,3]}, format='dict_of_lists
↪')
sage: list(G.out_branchings(1))
[Digraph on 4 vertices,
 Digraph on 4 vertices,
 Digraph on 4 vertices,
 Digraph on 4 vertices]
```

With the Petersen graph turned into a symmetric directed graph:

```
sage: G = graphs.PetersenGraph().to_directed()
sage: len(list(G.out_branchings(0)))
2000
```

With a non connected `DiGraph` and `spanning = True`:

```
sage: G = graphs.PetersenGraph().to_directed() + graphs.PetersenGraph().to_
↪directed()
sage: G.out_branchings(0, spanning=True)
Traceback (most recent call last):
...
ValueError: no spanning out branching from vertex (0) exist
```

With a non connected `DiGraph` and `spanning = False`:

```
sage: g=DiGraph([(0,1), (0,1), (1,2), (3,4)],multiedges=True)
sage: list(g.out_branchings(0, spanning=False))
[Digraph on 3 vertices, Digraph on 3 vertices]
```

With multiedges:

```
sage: G = DiGraph({0:[1,1,1], 1:[2,2]}, format='dict_of_lists',␣
↪multiedges=True)
sage: len(list(G.out_branchings(0)))
6
```

With a DiGraph already being a spanning out branching:

```
sage: G = DiGraph({0:[1,2], 1:[3,4], 2:[5], 3:[], 4:[], 5:[]}, format='dict_
↪of_lists')
sage: next(G.out_branchings(0)) == G
True
```

out_degree(*vertices=None*, *labels=False*)
Same as degree, but for out degree.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.out_degree(vertices=[0, 1 ,2], labels=True)
{0: 3, 1: 2, 2: 1}
sage: D.out_degree()
[3, 2, 1, 1, 2, 1]
sage: D.out_degree(2)
1
```

out_degree_iterator(*vertices=None*, *labels=False*)

Same as degree_iterator, but for out degree.

EXAMPLES:

```
sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: sorted(D.out_degree_iterator())
[2, 2, 2, 2, 3, 3, 3, 3]
sage: sorted(D.out_degree_iterator(labels=True))
[((0, 0), 2),
 ((0, 1), 3),
 ((0, 2), 3),
 ((0, 3), 2),
 ((1, 0), 2),
 ((1, 1), 3),
 ((1, 2), 3),
 ((1, 3), 2)]
```

out_degree_sequence()

Return the outdegree sequence of this digraph.

EXAMPLES:

The outdegree sequences of two digraphs:

```
sage: g = DiGraph({1: [2, 5, 6], 2: [3, 6], 3: [4, 6], 4: [6], 5: [4, 6]})
sage: g.out_degree_sequence()
[3, 2, 2, 2, 1, 0]
```

```
sage: V = [2, 3, 5, 7, 8, 9, 10, 11]
sage: E = [[], [8, 10], [11], [8, 11], [9], [], [], [2, 9, 10]]
sage: g = DiGraph(dict(zip(V, E)))
sage: g.out_degree_sequence()
[3, 2, 2, 1, 1, 0, 0, 0]
```

outgoing_edge_iterator(*vertices*, *labels=True*)

Return an iterator over all departing edges from vertices.

INPUT:

- `vertices` – a vertex or a list of vertices
- `labels` – boolean (default: `True`); whether to return edges as pairs of vertices, or as triples containing the labels.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: for a in D.outgoing_edge_iterator([0]):
....:     print(a)
```

(continues on next page)

(continued from previous page)

```
(0, 1, None)
(0, 2, None)
(0, 3, None)
```

outgoing_edges(*vertices*, *labels=True*)

Return a list of edges departing from vertices.

INPUT:

- `vertices` – a vertex or a list of vertices
- `labels` – boolean (default: `True`); whether to return edges as pairs of vertices, or as triples containing the labels.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.outgoing_edges([0])
[(0, 1, None), (0, 2, None), (0, 3, None)]
```

path_semigroup()

The partial semigroup formed by the paths of this quiver.

EXAMPLES:

```
sage: Q = DiGraph({1: {2: ['a', 'c']}, 2: {3: ['b']}})
sage: F = Q.path_semigroup(); F
Partial semigroup formed by the directed paths of Multi-digraph on 3 vertices
sage: list(F)
[e_1, e_2, e_3, a, c, b, a*b, c*b]
```

period()

Return the period of the current `DiGraph`.

The period of a directed graph is the largest integer that divides the length of every cycle in the graph. See the Wikipedia article Aperiodic_graph for more information.

EXAMPLES:

The following graph has period `2`:

```
sage: g = DiGraph({0: [1], 1: [0]})
sage: g.period()
2
```

The following graph has a cycle of length 2 and a cycle of length 3, so it has period `1`:

```
sage: g = DiGraph({0: [1, 4], 1: [2], 2: [0], 4: [0]})
sage: g.period()
1
```

Here is an example of computing the period of a digraph which is not strongly connected. By definition, it is the `gcd()` of the periods of its strongly connected components:

```
sage: g = DiGraph({-1: [-2], -2: [-3], -3: [-1],
....:      1: [2], 2: [1]})
sage: g.period()
1
sage: sorted([s.period() for s
```

(continues on next page)

(continued from previous page)

```
....:          in g.strongly_connected_components_subgraphs()])
[2, 3]
```

ALGORITHM:

See the networkX implementation of `is_aperiodic`, that is based on breadth first search.

See also:

is_aperiodic()

reverse()

Return a copy of digraph with edges reversed in direction.

EXAMPLES:

```
sage: D = DiGraph({0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1]})
sage: D.reverse()
Reverse of (): Digraph on 6 vertices
```

reverse_edge(*u*, *v=None*, *label=None*, *inplace=True*, *multiedges=None*)

Reverse the edge from u to v.

INPUT:

- `inplace` – boolean (default: `True`); if `False`, a new digraph is created and returned as output, otherwise `self` is modified.
- `multiedges` – boolean (default: `None`); how to decide what should be done in case of doubt (for instance when edge $(1, 2)$ is to be reversed in a graph while $(2, 1)$ already exists):
  - If set to `True`, input graph will be forced to allow parallel edges if necessary and edge $(1, 2)$ will appear twice in the graph.
  - If set to `False`, only one edge $(1, 2)$ will remain in the graph after $(2, 1)$ is reversed. Besides, the label of edge $(1, 2)$ will be overwritten with the label of edge $(2, 1)$.

  The default behaviour (`multiedges = None`) will raise an exception each time a subjective decision (setting `multiedges` to `True` or `False`) is necessary to perform the operation.

The following forms are all accepted:

- D.reverse_edge(1, 2)
- D.reverse_edge((1, 2))
- D.reverse_edge([1, 2])
- D.reverse_edge(1, 2, 'label')
- D.reverse_edge((1, 2, 'label'))
- D.reverse_edge([1, 2, 'label'])
- D.reverse_edge((1, 2), label='label')

EXAMPLES:

If `inplace` is `True` (default value), `self` is modified:

```
sage: D = DiGraph([(0, 1 ,2)])
sage: D.reverse_edge(0, 1)
sage: D.edges()
[(1, 0, 2)]
```

If `inplace` is `False`, `self` is not modified and a new digraph is returned:

```
sage: D = DiGraph([(0, 1, 2)])
sage: re = D.reverse_edge(0, 1, inplace=False)
sage: re.edges()
[(1, 0, 2)]
sage: D.edges()
[(0, 1, 2)]
```

If `multiedges` is `True`, `self` will be forced to allow parallel edges when and only when it is necessary:

```
sage: D = DiGraph([(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)])
sage: D.reverse_edge(1, 2, multiedges=True)
sage: D.edges()
[(2, 1, 'A'), (2, 1, 'A'), (2, 3, None)]
sage: D.allows_multiple_edges()
True
```

Even if `multiedges` is `True`, `self` will not be forced to allow parallel edges when it is not necessary:

```
sage: D = DiGraph( [(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)] )
sage: D.reverse_edge(2, 3, multiedges=True)
sage: D.edges()
[(1, 2, 'A'), (2, 1, 'A'), (3, 2, None)]
sage: D.allows_multiple_edges()
False
```

If user specifies `multiedges = False`, `self` will not be forced to allow parallel edges and a parallel edge will get deleted:

```
sage: D = DiGraph( [(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)] )
sage: D.edges()
[(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)]
sage: D.reverse_edge(1, 2, multiedges=False)
sage: D.edges()
[(2, 1, 'A'), (2, 3, None)]
```

Note that in the following graph, specifying `multiedges = False` will result in overwriting the label of $(1, 2)$ with the label of $(2, 1)$:

```
sage: D = DiGraph( [(1, 2, 'B'), (2, 1, 'A'), (2, 3, None)] )
sage: D.edges()
[(1, 2, 'B'), (2, 1, 'A'), (2, 3, None)]
sage: D.reverse_edge(2, 1, multiedges=False)
sage: D.edges()
[(1, 2, 'A'), (2, 3, None)]
```

If input edge in digraph has weight/label, then the weight/label should be preserved in the output digraph. User does not need to specify the weight/label when calling function:

```
sage: D = DiGraph([[0, 1, 2], [1, 2, 1]], weighted=True)
sage: D.reverse_edge(0, 1)
sage: D.edges()
[(1, 0, 2), (1, 2, 1)]
sage: re = D.reverse_edge([1, 2], inplace=False)
sage: re.edges()
[(1, 0, 2), (2, 1, 1)]
```

If `self` has multiple copies (parallel edges) of the input edge, only 1 of the parallel edges is reversed:

```
sage: D = DiGraph([(0, 1, '01'), (0, 1, '01'), (0, 1, 'cat'), (1, 2, '12')],␣
↪weighted=True, multiedges=True)
sage: re = D.reverse_edge([0, 1, '01'], inplace=False)
sage: re.edges()
[(0, 1, '01'), (0, 1, 'cat'), (1, 0, '01'), (1, 2, '12')]
```

If `self` has multiple copies (parallel edges) of the input edge but with distinct labels and no input label is specified, only 1 of the parallel edges is reversed (the edge that is labeled by the first label on the list returned by *edge_label()*):

```
sage: D = DiGraph([(0, 1, 'A'), (0, 1, 'B'), (0, 1, 'mouse'), (0, 1, 'cat')],␣
↪multiedges=true)
sage: D.edge_label(0, 1)
['cat', 'mouse', 'B', 'A']
sage: D.reverse_edge(0, 1)
sage: D.edges()
[(0, 1, 'A'), (0, 1, 'B'), (0, 1, 'mouse'), (1, 0, 'cat')]
```

Finally, an exception is raised when Sage does not know how to choose between allowing multiple edges and losing some data:

```
sage: D = DiGraph([(0, 1, 'A'), (1, 0, 'B')])
sage: D.reverse_edge(0, 1)
Traceback (most recent call last):
...
ValueError: reversing the given edge is about to create two parallel
edges but input digraph doesn't allow them - User needs to specify
multiedges is True or False.
```

The following syntax is supported, but note that you must use the `label` keyword:

```
sage: D = DiGraph()
sage: D.add_edge((1, 2), label='label')
sage: D.edges()
[(1, 2, 'label')]
sage: D.reverse_edge((1, 2), label='label')
sage: D.edges()
[(2, 1, 'label')]
sage: D.add_edge((1, 2), 'label')
sage: D.edges(sort=False)
[((1, 2), 'label', None), (2, 1, 'label')]
sage: D.reverse_edge((1, 2), 'label')
sage: D.edges(sort=False)
[('label', (1, 2), None), (2, 1, 'label')]
```

reverse_edges (*edges*, *inplace=True*, *multiedges=None*)

Reverse a list of edges.

INPUT:

- `edges` – a list of edges in the DiGraph.
- `inplace` – boolean (default: `True`); if `False`, a new digraph is created and returned as output, otherwise `self` is modified.
- `multiedges` – boolean (default: `None`); if `True`, input graph will be forced to allow parallel edges when necessary (for more information see the documentation of *reverse_edge()*)

See also:

reverse_edge() - Reverses a single edge.

EXAMPLES:

If `inplace` is `True` (default value), `self` is modified:

```
sage: D = DiGraph({ 0: [1, 1, 3], 2: [3, 3], 4: [1, 5]}, multiedges=true)
sage: D.reverse_edges([[0, 1], [0, 3]])
sage: D.reverse_edges([(2, 3), (4, 5)])
sage: D.edges()
[(0, 1, None), (1, 0, None), (2, 3, None), (3, 0, None),
 (3, 2, None), (4, 1, None), (5, 4, None)]
```

If `inplace` is `False`, `self` is not modified and a new digraph is returned:

```
sage: D = DiGraph([(0, 1, 'A'), (1, 0, 'B'), (1, 2, 'C')])
sage: re = D.reverse_edges([(0, 1), (1, 2)],
....:                      inplace=False,
....:                      multiedges=True)
sage: re.edges()
[(1, 0, 'A'), (1, 0, 'B'), (2, 1, 'C')]
sage: D.edges()
[(0, 1, 'A'), (1, 0, 'B'), (1, 2, 'C')]
sage: D.allows_multiple_edges()
False
sage: re.allows_multiple_edges()
True
```

If `multiedges` is `True`, `self` will be forced to allow parallel edges when and only when it is necessary:

```
sage: D = DiGraph([(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)])
sage: D.reverse_edges([(1, 2), (2, 3)], multiedges=True)
sage: D.edges()
[(2, 1, 'A'), (2, 1, 'A'), (3, 2, None)]
sage: D.allows_multiple_edges()
True
```

Even if `multiedges` is `True`, `self` will not be forced to allow parallel edges when it is not necessary:

```
sage: D = DiGraph([(1, 2, 'A'), (2, 1, 'A'), (2, 3, None)])
sage: D.reverse_edges([(2, 3)], multiedges=True)
sage: D.edges()
[(1, 2, 'A'), (2, 1, 'A'), (3, 2, None)]
sage: D.allows_multiple_edges()
False
```

If `multiedges` is `False`, `self` will not be forced to allow parallel edges and an edge will get deleted:

```
sage: D = DiGraph([(1, 2), (2, 1)])
sage: D.edges()
[(1, 2, None), (2, 1, None)]
sage: D.reverse_edges([(1, 2)], multiedges=False)
sage: D.edges()
[(2, 1, None)]
```

If input edge in digraph has weight/label, then the weight/label should be preserved in the output digraph. User does not need to specify the weight/label when calling function:

```
sage: D = DiGraph([(0, 1, '01'), (1, 2, 1), (2, 3, '23')], weighted=True)
sage: D.reverse_edges([(0, 1, '01'), (1, 2), (2, 3)])
sage: D.edges()
[(1, 0, '01'), (2, 1, 1), (3, 2, '23')]
```

sinks()

Return a list of sinks of the digraph.

OUTPUT:

- list of the vertices of the digraph that have no edges beginning at them

EXAMPLES:

```
sage: G = DiGraph({1: {3: ['a']}, 2: {3: ['b']}})
sage: G.sinks()
[3]
sage: T = DiGraph({1: {}})
sage: T.sinks()
[1]
```

sources()

Return a list of sources of the digraph.

OUTPUT:

- list of the vertices of the digraph that have no edges going into them

EXAMPLES:

```
sage: G = DiGraph({1: {3: ['a']}, 2: {3: ['b']}})
sage: G.sources()
[1, 2]
sage: T = DiGraph({1: {}})
sage: T.sources()
[1]
```

strong_articulation_points(G)

Return the strong articulation points of this digraph.

A vertex is a strong articulation point if its deletion increases the number of strongly connected components. This method implements the algorithm described in [ILS2012]. The time complexity is dominated by the time complexity of the immediate dominators finding algorithm.

OUTPUT: The list of strong articulation points.

EXAMPLES:

Two cliques sharing a vertex:

```
sage: from sage.graphs.connectivity import strong_articulation_points
sage: D = digraphs.Complete(4)
sage: D.add_clique([3, 4, 5, 6])
sage: strong_articulation_points(D)
[3]
sage: D.strong_articulation_points()
[3]
```

Two cliques connected by some arcs:

```
sage: D = digraphs.Complete(4) * 2
sage: D.add_edges([(0, 4), (7, 3)])
sage: sorted(strong_articulation_points(D))
[0, 3, 4, 7]
sage: D.add_edge(1, 5)
sage: sorted(strong_articulation_points(D))
[3, 7]
sage: D.add_edge(6, 2)
sage: strong_articulation_points(D)
[]
```

See also:

- *strongly_connected_components()*
- *dominator_tree()*

strongly_connected_component_containing_vertex(*G*, *v*)

Return the strongly connected component containing a given vertex

INPUT:

- G – the input DiGraph
- v – a vertex

EXAMPLES:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: from sage.graphs.connectivity import strongly_connected_component_
↪containing_vertex
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: strongly_connected_component_containing_vertex(d, 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: d.strongly_connected_component_containing_vertex(0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
sage: g = DiGraph([(0, 1), (1, 0), (1, 2), (2, 3), (3, 2)])
sage: strongly_connected_component_containing_vertex(g, 0)
[0, 1]
```

strongly_connected_components(*G*)

Return the lists of vertices in each strongly connected components (SCCs).

This method implements the Tarjan algorithm to compute the strongly connected components of the digraph. It returns a list of lists of vertices, each list of vertices representing a strongly connected component.

The basic idea of the algorithm is this: a depth-first search (DFS) begins from an arbitrary start node (and subsequent DFSes are conducted on any nodes that have not yet been found). As usual with DFSes, the search visits every node of the graph exactly once, declining to revisit any node that has already been explored. Thus, the collection of search trees is a spanning forest of the graph. The strongly connected components correspond to the subtrees of this spanning forest that have no edge directed outside the subtree.

To recover these components, during the DFS, we keep the index of a node, that is, the position in the DFS tree, and the lowlink: as soon as the subtree rooted at v has been fully explored, the lowlink of v is the smallest index reachable from v passing from descendants of v. If the subtree rooted at v has been fully explored, and the index of v equals the lowlink of v, that whole subtree is a new SCC.

For more information, see the Wikipedia article Tarjan's_strongly_connected_components_algorithm.

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import tarjan_strongly_
↪connected_components
sage: tarjan_strongly_connected_components(digraphs.Path(3))
[[2], [1], [0]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.strongly_connected_components()
[[3], [2], [1], [0], [6], [5], [4]]
sage: D.add_edge([2,0])
sage: D.strongly_connected_components()
[[3], [0, 1, 2], [6], [5], [4]]
sage: D = DiGraph([('a','b'), ('b','c'), ('c', 'd'), ('d', 'b'), ('c', 'e')])
sage: [sorted(scc) for scc in D.strongly_connected_components()]
[['e'], ['b', 'c', 'd'], ['a']]
```

strongly_connected_components_digraph(*G*, *keep_labels=False*)

Return the digraph of the strongly connected components

The digraph of the strongly connected components of a graph G has a vertex per strongly connected component included in G. There is an edge from a component C_1 to a component C_2 if there is an edge in G from a vertex $u_1 \in C_1$ to a vertex $u_2 \in C_2$.

INPUT:

- `G` – the input DiGraph
- `keep_labels` – boolean (default: `False`); when `keep_labels=True`, the resulting digraph has an edge from a component C_i to a component C_j for each edge in G from a vertex $u_i \in C_i$ to a vertex $u_j \in C_j$. Hence the resulting digraph may have loops and multiple edges. However, edges in the result with same source, target, and label are not duplicated (see examples below). When `keep_labels=False`, the return digraph is simple, so without loops nor multiple edges, and edges are unlabelled.

EXAMPLES:

Such a digraph is always acyclic:

```
sage: from sage.graphs.connectivity import strongly_connected_components_
↪digraph
sage: g = digraphs.RandomDirectedGNP(15, .1)
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: scc_digraph.is_directed_acyclic()
True
sage: scc_digraph = g.strongly_connected_components_digraph()
sage: scc_digraph.is_directed_acyclic()
True
```

The vertices of the digraph of strongly connected components are exactly the strongly connected components:

```
sage: g = digraphs.ButterflyGraph(2)
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: g.is_directed_acyclic()
True
```

(continues on next page)

(continued from previous page)

```
sage: V_scc = list(scc_digraph)
sage: all(Set(scc) in V_scc for scc in g.strongly_connected_components())
True
```

The following digraph has three strongly connected components, and the digraph of those is a `TransitiveTournament()`:

```
sage: g = DiGraph({0: {1: "01", 2: "02", 3: "03"}, 1: {2: "12"}, 2:{1: "21",␣
↪3: "23"}})
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: scc_digraph.is_isomorphic(digraphs.TransitiveTournament(3))
True
```

By default, the labels are discarded, and the result has no loops nor multiple edges. If `keep_labels` is `True`, then the labels are kept, and the result is a multi digraph, possibly with multiple edges and loops. However, edges in the result with same source, target, and label are not duplicated (see the edges from 0 to the strongly connected component $\{1, 2\}$ below):

```
sage: g = DiGraph({0: {1: "0-12", 2: "0-12", 3: "0-3"}, 1: {2: "1-2", 3: "1-3
↪"}, 2: {1: "2-1", 3: "2-3"}})
sage: g.order(), g.size()
(4, 7)
sage: scc_digraph = strongly_connected_components_digraph(g, keep_labels=True)
sage: (scc_digraph.order(), scc_digraph.size())
(3, 6)
sage: set(g.edge_labels()) == set(scc_digraph.edge_labels())
True
```

strongly_connected_components_subgraphs(G)

Return the strongly connected components as a list of subgraphs.

EXAMPLES:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: from sage.graphs.connectivity import strongly_connected_components_
↪subgraphs
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: strongly_connected_components_subgraphs(d)
[Subgraph of (Petersen graph): Digraph on 10 vertices]
sage: d.strongly_connected_components_subgraphs()
[Subgraph of (Petersen graph): Digraph on 10 vertices]
```

```
sage: g = DiGraph([(0, 1), (1, 0), (1, 2), (2, 3), (3, 2)])
sage: strongly_connected_components_subgraphs(g)
[Subgraph of (): Digraph on 2 vertices, Subgraph of (): Digraph on 2 vertices]
```

to_directed()

Since the graph is already directed, simply returns a copy of itself.

EXAMPLES:

```
sage: DiGraph({0: [1, 2, 3], 4: [5, 1]}).to_directed()
Digraph on 6 vertices
```

to_undirected(*data_structure=None*, *sparse=None*)

Return an undirected version of the graph.

Every directed edge becomes an edge.

INPUT:

- `data_structure` – string (default: `None`); one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of *Graph* or *DiGraph*.
- `sparse` – boolean (default: `None`); `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`.

EXAMPLES:

```
sage: D = DiGraph({0: [1, 2], 1: [0]})
sage: G = D.to_undirected()
sage: D.edges(labels=False)
[(0, 1), (0, 2), (1, 0)]
sage: G.edges(labels=False)
[(0, 1), (0, 2)]
```

topological_sort(*implementation='default'*)

Return a topological sort of the digraph if it is acyclic.

If the digraph contains a directed cycle, a `TypeError` is raised. As topological sorts are not necessarily unique, different implementations may yield different results.

A topological sort is an ordering of the vertices of the digraph such that each vertex comes before all of its successors. That is, if u comes before v in the sort, then there may be a directed path from u to v, but there will be no directed path from v to u.

INPUT:

- `implementation` – string (default: `"default"`); either use the default Cython implementation, or the default NetworkX library (`implementation = "NetworkX"`)

See also:

- *is_directed_acyclic()* – Tests whether a directed graph is acyclic (can also join a certificate – a topological sort or a circuit in the graph).

EXAMPLES:

```
sage: D = DiGraph({0: [1, 2, 3], 4: [2, 5], 1: [8], 2: [7], 3: [7],
....:   5: [6, 7], 7: [8], 6: [9], 8: [10], 9: [10]})
sage: D.plot(layout='circular').show()
sage: D.topological_sort()
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]
```

```
sage: D.add_edge(9, 7)
sage: D.topological_sort()
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]
```

Using the NetworkX implementation

```
sage: list(D.topological_sort(implementation="NetworkX"))
[4, 5, 6, 9, 0, 3, 2, 7, 1, 8, 10]
```

```
sage: D.add_edge(7, 4)
sage: D.topological_sort()
Traceback (most recent call last):
...
TypeError: digraph is not acyclic; there is no topological sort
```

topological_sort_generator()

Return an iterator over all topological sorts of the digraph if it is acyclic.

If the digraph contains a directed cycle, a `TypeError` is raised.

A topological sort is an ordering of the vertices of the digraph such that each vertex comes before all of its successors. That is, if u comes before v in the sort, then there may be a directed path from u to v, but there will be no directed path from v to u. See also *topological_sort()*.

AUTHORS:

- Mike Hansen - original implementation
- Robert L. Miller: wrapping, documentation

REFERENCE:

- [1] Pruesse, Gara and Ruskey, Frank. Generating Linear Extensions Fast. SIAM J. Comput., Vol. 23 (1994), no. 2, pp. 373-386.

EXAMPLES:

```
sage: D = DiGraph({0: [1, 2], 1: [3], 2: [3, 4]})
sage: D.plot(layout='circular').show()
sage: list(D.topological_sort_generator())
[[0, 1, 2, 3, 4], [0, 2, 1, 3, 4], [0, 2, 1, 4, 3], [0, 2, 4, 1, 3], [0, 1, 2,
↪ 4, 3]]
```

```
sage: for sort in D.topological_sort_generator():
....:     for u, v in D.edge_iterator(labels=False):
....:         if sort.index(u) > sort.index(v):
....:             print("this should never happen")
```

1.4 Bipartite graphs

This module implements bipartite graphs.

AUTHORS:

- Robert L. Miller (2008-01-20): initial version
- Ryan W. Hinton (2010-03-04): overrides for adding and deleting vertices and edges

class sage.graphs.bipartite_graph.**BipartiteGraph**(*data=None*, *partition=None*, *check=True*, **args*, ***kwds*)

Bases: *sage.graphs.graph.Graph*

Bipartite graph.

INPUT:

- `data` – can be any of the following:
  1. Empty or `None` (creates an empty graph).

2. An arbitrary graph.
3. A reduced adjacency matrix.

   A reduced adjacency matrix contains only the non-redundant portion of the full adjacency matrix for the bipartite graph. Specifically, for zero matrices of the appropriate size, for the reduced adjacency matrix `H`, the full adjacency matrix is `[[0, H'], [H, 0]]`. The columns correspond to vertices on the left, and the rows correspond to vertices on the right.
4. A file in alist format.

   The alist file format is described at http://www.inference.phy.cam.ac.uk/mackay/codes/alist.html
5. From a NetworkX bipartite graph.

- `partition` – (default: `None`); a tuple defining vertices of the left and right partition of the graph. Partitions will be determined automatically if `partition` is `None`.
- `check` – boolean (default: `True`); if `True`, an invalid input partition raises an exception. In the other case offending edges simply won't be included.
- `loops` – ignored; bipartite graphs cannot have loops
- `multiedges` – boolean (default: `None`); whether to allow multiple edges
- `weighted` – boolean (default: `None`); whether graph thinks of itself as weighted or not. See `self.weighted()`

Note: All remaining arguments are passed to the `Graph` constructor

EXAMPLES:

1. No inputs or `None` for the input creates an empty graph:

```
sage: B = BipartiteGraph()
sage: type(B)
<class 'sage.graphs.bipartite_graph.BipartiteGraph'>
sage: B.order()
0
sage: B == BipartiteGraph(None)
True
```

2. From a graph: without any more information, finds a bipartition:

```
sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B = BipartiteGraph(graphs.CycleGraph(5))
Traceback (most recent call last):
...
ValueError: input graph is not bipartite
sage: G = Graph({0: [5, 6], 1: [4, 5], 2: [4, 6], 3: [4, 5, 6]})
sage: B = BipartiteGraph(G)
sage: B == G
True
sage: B.left
{0, 1, 2, 3}
sage: B.right
{4, 5, 6}
sage: B = BipartiteGraph({0: [5, 6], 1: [4, 5], 2: [4, 6], 3: [4, 5, 6]})
sage: B == G
True
```

(continues on next page)

(continued from previous page)

```
sage: B.left
{0, 1, 2, 3}
sage: B.right
{4, 5, 6}
```

3. If a `Graph` or `DiGraph` is used as data, you can specify a partition using `partition` argument. Note that if such graph is not bipartite, then Sage will raise an error. However, if one specifies `check=False`, the offending edges are simply deleted (along with those vertices not appearing in either list). We also lump creating one bipartite graph from another into this category:

```
sage: P = graphs.PetersenGraph()
sage: partition = [list(range(5)), list(range(5, 10))]
sage: B = BipartiteGraph(P, partition)
Traceback (most recent call last):
...
TypeError: input graph is not bipartite with respect to the given partition

sage: B = BipartiteGraph(P, partition, check=False)
sage: B.left
{0, 1, 2, 3, 4}
sage: B.show()
```

```
sage: G = Graph({0: [5, 6], 1: [4, 5], 2: [4, 6], 3: [4, 5, 6]})
sage: B = BipartiteGraph(G)
sage: B2 = BipartiteGraph(B)
sage: B == B2
True
sage: B3 = BipartiteGraph(G, [list(range(4)), list(range(4, 7))])
sage: B3
Bipartite graph on 7 vertices
sage: B3 == B2
True
```

```
sage: G = Graph({0: [], 1: [], 2: []})
sage: part = (list(range(2)), [2])
sage: B = BipartiteGraph(G, part)
sage: B2 = BipartiteGraph(B)
sage: B == B2
True
```

```
sage: d = DiGraph(6)
sage: d.add_edge(0, 1)
sage: part=[[1, 2, 3], [0, 4, 5]]
sage: b = BipartiteGraph(d, part)
sage: b.left
{1, 2, 3}
sage: b.right
{0, 4, 5}
```

4. From a reduced adjacency matrix:

```
sage: M = Matrix([(1,1,1,0,0,0,0), (1,0,0,1,1,0,0),
....:             (0,1,0,1,0,1,0), (1,1,0,1,0,0,1)])
sage: M
[1 1 1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[1 0 0 1 1 0 0]
[0 1 0 1 0 1 0]
[1 1 0 1 0 0 1]
sage: H = BipartiteGraph(M); H
Bipartite graph on 11 vertices
sage: H.edges()
[(0, 7, None),
 (0, 8, None),
 (0, 10, None),
 (1, 7, None),
 (1, 9, None),
 (1, 10, None),
 (2, 7, None),
 (3, 8, None),
 (3, 9, None),
 (3, 10, None),
 (4, 8, None),
 (5, 9, None),
 (6, 10, None)]
```

```
sage: M = Matrix([(1, 1, 2, 0, 0), (0, 2, 1, 1, 1), (0, 1, 2, 1, 1)])
sage: B = BipartiteGraph(M, multiedges=True, sparse=True)
sage: B.edges()
[(0, 5, None),
 (1, 5, None),
 (1, 6, None),
 (1, 6, None),
 (1, 7, None),
 (2, 5, None),
 (2, 5, None),
 (2, 6, None),
 (2, 7, None),
 (2, 7, None),
 (3, 6, None),
 (3, 7, None),
 (4, 6, None),
 (4, 7, None)]
```

```
sage: F.<a> = GF(4)
sage: MS = MatrixSpace(F, 2, 3)
sage: M = MS.matrix([[0, 1, a + 1], [a, 1, 1]])
sage: B = BipartiteGraph(M, weighted=True, sparse=True)
sage: B.edges()
[(0, 4, a), (1, 3, 1), (1, 4, 1), (2, 3, a + 1), (2, 4, 1)]
sage: B.weighted()
True
```

5. From an alist file:

```
sage: file_name = os.path.join(SAGE_TMP, 'deleteme.alist.txt')
sage: fi = open(file_name, 'w')
sage: _ = fi.write("7 4 \n 3 4 \n 3 3 1 3 1 1 1 \n 3 3 3 4 \n\
                    1 2 4 \n 1 3 4 \n 1 0 0 \n 2 3 4 \n\
                    2 0 0 \n 3 0 0 \n 4 0 0 \n\
                    1 2 3 0 \n 1 4 5 0 \n 2 4 6 0 \n 1 2 4 7 \n")
sage: fi.close()
```

(continues on next page)

(continued from previous page)

```
sage: B = BipartiteGraph(file_name)
sage: B.is_isomorphic(H)
True
```

6. From a NetworkX bipartite graph:

```
sage: import networkx
sage: G = graphs.OctahedralGraph()
sage: N = networkx.make_clique_bipartite(G.networkx_graph())
sage: B = BipartiteGraph(N)
```

add_edge(*u*, *v=None*, *label=None*)
Add an edge from u to v.

INPUT:

- `u` – the tail of an edge.
- `v` – (default: `None`); the head of an edge. If `v=None`, then attempt to understand `u` as a edge tuple.
- `label` – (default: `None`); the label of the edge `(u, v)`.

The following forms are all accepted:

- `G.add_edge(1, 2)`
- `G.add_edge((1, 2))`
- `G.add_edges([(1, 2)])`
- `G.add_edge(1, 2, 'label')`
- `G.add_edge((1, 2, 'label'))`
- `G.add_edges([(1, 2, 'label')])`

See `add_edge()` for more detail.

This method simply checks that the edge endpoints are in different partitions. If a new vertex is to be created, it will be added to the proper partition. If both vertices are created, the first one will be added to the left partition, the second to the right partition.

add_edges(*edges*, *loops=True*)
Add edges from an iterable container.

INPUT:

- `edges` – an iterable of edges, given either as `(u, v)` or `(u, v, label)`.
- `loops` – ignored

See `add_edges()` for more detail.

This method simply checks that the edge endpoints are in different partitions. If a new vertex is to be created, it will be added to the proper partition. If both vertices are created, the first one will be added to the left partition, the second to the right partition.

EXAMPLES:

```
sage: bg = BipartiteGraph()
sage: bg.add_vertices([0, 1, 2], left=[True, False, True])
sage: bg.add_edges([(0, 1), (2, 1)])
sage: bg.add_edges([[0, 2]])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
RuntimeError: edge vertices must lie in different partitions
```

Loops will raise an error:

```
sage: bg.add_edges([[0, 3], [3, 3]])
Traceback (most recent call last):
...
RuntimeError: edge vertices must lie in different partitions
```

add_vertex(*name=None*, *left=False*, *right=False*)

Create an isolated vertex. If the vertex already exists, then nothing is done.

INPUT:

- `name` – (default: `None`); name of the new vertex. If no name is specified, then the vertex will be represented by the least non-negative integer not already representing a vertex. Name must be an immutable object and cannot be `None`.
- `left` – boolean (default: `False`); if `True`, puts the new vertex in the left partition.
- `right` – boolean (default: `False`); if `True`, puts the new vertex in the right partition.

Obviously, `left` and `right` are mutually exclusive.

As it is implemented now, if a graph G has a large number of vertices with numeric labels, then `G.add_vertex()` could potentially be slow, if name is `None`.

OUTPUT:

- If `name` is `None`, the new vertex name is returned. `None` otherwise.

EXAMPLES:

```
sage: G = BipartiteGraph()
sage: G.add_vertex(left=True)
0
sage: G.add_vertex(right=True)
1
sage: G.vertices()
[0, 1]
sage: G.left
{0}
sage: G.right
{1}
```

add_vertices(*vertices*, *left=False*, *right=False*)

Add vertices to the bipartite graph from an iterable container of vertices.

Vertices that already exist in the graph will not be added again.

INPUT:

- `vertices` – sequence of vertices to add.
- `left` – (default: `False`); either `True` or sequence of same length as `vertices` with `True`/`False` elements.
- `right` – (default: `False`); either `True` or sequence of the same length as `vertices` with `True`/`False` elements.

Only one of `left` and `right` keywords should be provided. See the examples below.

EXAMPLES:

```
sage: bg = BipartiteGraph()
sage: bg.add_vertices([0, 1, 2], left=True)
sage: bg.add_vertices([3, 4, 5], left=[True, False, True])
sage: bg.add_vertices([6, 7, 8], right=[True, False, True])
sage: bg.add_vertices([9, 10, 11], right=True)
sage: bg.left
{0, 1, 2, 3, 5, 7}
sage: bg.right
{4, 6, 8, 9, 10, 11}
```

allow_loops(*new*, *check=True*)

Change whether loops are permitted in the (di)graph

Note: This method overwrite the *allow_loops()* method to ensure that loops are forbidden in *BipartiteGraph*.

INPUT:

- `new` – boolean

EXAMPLES:

```
sage: B = BipartiteGraph()
sage: B.allow_loops(True)
Traceback (most recent call last):
...
ValueError: loops are not allowed in bipartite graphs
```

bipartition()

Return the underlying bipartition of the bipartite graph.

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B.bipartition()
({0, 2}, {1, 3})
```

complement()

Return a complement of this graph.

EXAMPLES:

```
sage: B = BipartiteGraph({1: [2, 4], 3: [4, 5]})
sage: G = B.complement(); G
Graph on 5 vertices
sage: G.edges(labels=False)
[(1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)]
```

delete_vertex(*vertex*, *in_order=False*)

Delete vertex, removing all incident edges.

Deleting a non-existent vertex will raise an exception.

INPUT:

- `vertex` – a vertex to delete.

- `in_order` – boolean (default `False`); if `True`, deletes the i-th vertex in the sorted list of vertices, i.e. `G.vertices()[i]`.

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B
Bipartite cycle graph: graph on 4 vertices
sage: B.delete_vertex(0)
sage: B
Bipartite cycle graph: graph on 3 vertices
sage: B.left
{2}
sage: B.edges()
[(1, 2, None), (2, 3, None)]
sage: B.delete_vertex(3)
sage: B.right
{1}
sage: B.edges()
[(1, 2, None)]
sage: B.delete_vertex(0)
Traceback (most recent call last):
...
ValueError: vertex (0) not in the graph
```

```
sage: g = Graph({'a': ['b'], 'c': ['b']})
sage: bg = BipartiteGraph(g)  # finds bipartition
sage: bg.vertices()
['a', 'b', 'c']
sage: bg.delete_vertex('a')
sage: bg.edges()
[('b', 'c', None)]
sage: bg.vertices()
['b', 'c']
sage: bg2 = BipartiteGraph(g)
sage: bg2.delete_vertex(0, in_order=True)
sage: bg2 == bg
True
```

delete_vertices(*vertices*)

Remove vertices from the bipartite graph taken from an iterable sequence of vertices.

Deleting a non-existent vertex will raise an exception.

INPUT:

- `vertices` – a sequence of vertices to remove

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B
Bipartite cycle graph: graph on 4 vertices
sage: B.delete_vertices([0, 3])
sage: B
Bipartite cycle graph: graph on 2 vertices
sage: B.left
{2}
sage: B.right
```

(continues on next page)

(continued from previous page)

```
{1}
sage: B.edges()
[(1, 2, None)]
sage: B.delete_vertices([0])
Traceback (most recent call last):
...
ValueError: vertex (0) not in the graph
```

is_bipartite(*certificate=False*)

Check whether the graph is bipartite.

This method always returns `True` as first value, plus a certificate when `certificate == True`.

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate. If set to `True`, the certificate returned is a proper 2-coloring of the vertices.

See also:

is_bipartite()

EXAMPLES:

```
sage: g = BipartiteGraph(graphs.RandomBipartite(3, 3, .5))
sage: g.is_bipartite()
True
sage: g.is_bipartite(certificate=True)  # random
(True, {(0, 0): 0, (0, 1): 0, (0, 2): 0, (1, 0): 1, (1, 1): 1, (1, 2): 1})
```

load_afile(*fname*)

Load into the current object the bipartite graph specified in the given file name.

This file should follow David MacKay's alist format, see http://www.inference.phy.cam.ac.uk/mackay/codes/data.html for examples and definition of the format.

EXAMPLES:

```
sage: file_name = os.path.join(SAGE_TMP, 'deleteme.alist.txt')
sage: fi = open(file_name, 'w')
sage: _ = fi.write("7 4 \n 3 4 \n 3 3 1 3 1 1 1 \n 3 3 3 4 \n\
                    1 2 4 \n 1 3 4 \n 1 0 0 \n 2 3 4 \n\
                    2 0 0 \n 3 0 0 \n 4 0 0 \n\
                    1 2 3 0 \n 1 4 5 0 \n 2 4 6 0 \n 1 2 4 7 \n")
sage: fi.close()
sage: B = BipartiteGraph()
sage: B.load_afile(file_name)
Bipartite graph on 11 vertices
sage: B.edges()
[(0, 7, None),
 (0, 8, None),
 (0, 10, None),
 (1, 7, None),
 (1, 9, None),
 (1, 10, None),
 (2, 7, None),
 (3, 8, None),
 (3, 9, None),
 (3, 10, None),
```

(continues on next page)

(continued from previous page)

```
 (4, 8, None),
 (5, 9, None),
 (6, 10, None)]
 sage: B2 = BipartiteGraph(file_name)
 sage: B2 == B
 True
```

matching(*value_only=False*, *algorithm=None*, *use_edge_labels=False*, *solver=None*, *verbose=0*)

Return a maximum matching of the graph represented by the list of its edges.

Given a graph G such that each edge e has a weight w_e, a maximum matching is a subset S of the edges of G of maximum weight such that no two edges of S are incident with each other.

INPUT:

- `value_only` – boolean (default: `False`); when set to `True`, only the cardinal (or the weight) of the matching is returned
- `algorithm` – string (default: `"Hopcroft-Karp"` if `use_edge_labels==False`, otherwise `"Edmonds"`); algorithm to use among:
  - `"Hopcroft-Karp"` selects the default bipartite graph algorithm as implemented in NetworkX
  - `"Eppstein"` selects Eppstein's algorithm as implemented in NetworkX
  - `"Edmonds"` selects Edmonds' algorithm as implemented in NetworkX
  - `"LP"` uses a Linear Program formulation of the matching problem
- `use_edge_labels` – boolean (default: `False`)
  - when set to `True`, computes a weighted matching where each edge is weighted by its label (if an edge has no label, 1 is assumed); only if `algorithm` is `"Edmonds"`, `"LP"`
  - when set to `False`, each edge has weight 1
- `solver` – (default: `None`) a specific Linear Program (LP) solver to be used
- `verbose` – integer (default: `0`); sets the level of verbosity: set to 0 by default, which means quiet

See also:

- Wikipedia article Matching_(graph_theory)
- *matching()*

EXAMPLES:

Maximum matching in a cycle graph:

```
sage: G = BipartiteGraph(graphs.CycleGraph(10))
sage: G.matching()
[(0, 1, None), (2, 3, None), (4, 5, None), (6, 7, None), (8, 9, None)]
```

The size of a maximum matching in a complete bipartite graph using Eppstein:

```
sage: G = BipartiteGraph(graphs.CompleteBipartiteGraph(4,5))
sage: G.matching(algorithm="Eppstein", value_only=True)
4
```

matching_polynomial(*algorithm='Godsil'*, *name=None*)

Compute the matching polynomial.

The *matching polynomial* is defined as in [God1993], where $p(G, k)$ denotes the number of k-matchings (matchings with k edges) in G :

$$\mu(x) = \sum_{k \geq 0} (-1)^k p(G, k) x^{n-2k}$$

INPUT:

- `algorithm` – string (default: `"Godsil"`); either "Godsil" or "rook"; "rook" is usually faster for larger graphs
- `name` – string (default: `None`); name of the variable in the polynomial, set to x when `name` is `None`

EXAMPLES:

```
sage: BipartiteGraph(graphs.CubeGraph(3)).matching_polynomial()
x^8 - 12*x^6 + 42*x^4 - 44*x^2 + 9
```

```
sage: x = polygen(QQ)
sage: g = BipartiteGraph(graphs.CompleteBipartiteGraph(16, 16))
sage: bool(factorial(16) * laguerre(16, x^2) == g.matching_
↪polynomial(algorithm='rook'))
True
```

Compute the matching polynomial of a line with 60 vertices:

```
sage: from sage.functions.orthogonal_polys import chebyshev_U
sage: g = next(graphs.trees(60))
sage: chebyshev_U(60, x/2) == BipartiteGraph(g).matching_polynomial(algorithm=
↪'rook')
True
```

The matching polynomial of a tree is equal to its characteristic polynomial:

```
sage: g = graphs.RandomTree(20)
sage: p = g.characteristic_polynomial()
sage: p == BipartiteGraph(g).matching_polynomial(algorithm='rook')
True
```

plot(**args*, ***kwds*)

Override Graph's plot function, to illustrate the bipartite nature.

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(20))
sage: B.plot()
Graphics object consisting of 41 graphics primitives
```

project_left()

Project `self` onto left vertices. Edges are 2-paths in the original.

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(20))
sage: G = B.project_left()
sage: G.order(), G.size()
(10, 10)
```

project_right()
Project `self` onto right vertices. Edges are 2-paths in the original.

EXAMPLES:

```
sage: B = BipartiteGraph(graphs.CycleGraph(20))
sage: G = B.project_right()
sage: G.order(), G.size()
(10, 10)
```

reduced_adjacency_matrix(*sparse=True*)
Return the reduced adjacency matrix for the given graph.

A reduced adjacency matrix contains only the non-redundant portion of the full adjacency matrix for the bipartite graph. Specifically, for zero matrices of the appropriate size, for the reduced adjacency matrix `H`, the full adjacency matrix is `[[0, H'], [H, 0]]`.

INPUT:

- `sparse` – boolean (default: `True`); whether to return a sparse matrix

EXAMPLES:

Bipartite graphs that are not weighted will return a matrix over ZZ:

```
sage: M = Matrix([(1,1,1,0,0,0,0), (1,0,0,1,1,0,0),
....:             (0,1,0,1,0,1,0), (1,1,0,1,0,0,1)])
sage: B = BipartiteGraph(M)
sage: N = B.reduced_adjacency_matrix()
sage: N
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[0 1 0 1 0 1 0]
[1 1 0 1 0 0 1]
sage: N == M
True
sage: N[0,0].parent()
Integer Ring
```

Multi-edge graphs also return a matrix over ZZ:

```
sage: M = Matrix([(1,1,2,0,0), (0,2,1,1,1), (0,1,2,1,1)])
sage: B = BipartiteGraph(M, multiedges=True, sparse=True)
sage: N = B.reduced_adjacency_matrix()
sage: N == M
True
sage: N[0,0].parent()
Integer Ring
```

Weighted graphs will return a matrix over the ring given by their (first) weights:

```
sage: F.<a> = GF(4)
sage: MS = MatrixSpace(F, 2, 3)
sage: M = MS.matrix([[0, 1, a+1], [a, 1, 1]])
sage: B = BipartiteGraph(M, weighted=True, sparse=True)
sage: N = B.reduced_adjacency_matrix(sparse=False)
sage: N == M
True
sage: N[0,0].parent()
Finite Field in a of size 2^2
```

save_afile(*fname*)
Save the graph to file in alist format.

Saves this graph to file in David MacKay's alist format, see http://www.inference.phy.cam.ac.uk/mackay/codes/data.html for examples and definition of the format.

EXAMPLES:

```
sage: M = Matrix([(1,1,1,0,0,0,0), (1,0,0,1,1,0,0),
....:             (0,1,0,1,0,1,0), (1,1,0,1,0,0,1)])
sage: M
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[0 1 0 1 0 1 0]
[1 1 0 1 0 0 1]
sage: b = BipartiteGraph(M)
sage: file_name = os.path.join(SAGE_TMP, 'deleteme.alist.txt')
sage: b.save_afile(file_name)
sage: b2 = BipartiteGraph(file_name)
sage: b.is_isomorphic(b2)
True
```

to_undirected()
Return an undirected Graph (without bipartite constraint) of the given object.

EXAMPLES:

```
sage: BipartiteGraph(graphs.CycleGraph(6)).to_undirected()
Cycle graph: Graph on 6 vertices
```

vertex_cover(*algorithm='Konig'*, *value_only=False*, *reduction_rules=True*, *solver=None*, *verbosity=0*)
Return a minimum vertex cover of self represented by a set of vertices.

A minimum vertex cover of a graph is a set S of vertices such that each edge is incident to at least one element of S, and such that S is of minimum cardinality. For more information, see Wikipedia article Vertex_cover.

Equivalently, a vertex cover is defined as the complement of an independent set.

As an optimization problem, it can be expressed as follows:

$$\begin{aligned}\text{Minimize : } & \sum_{v \in G} b_v \\ \text{Such that : } & \forall (u,v) \in G.edges(), b_u + b_v \geq 1 \\ & \forall x \in G, b_x \text{ is a binary variable}\end{aligned}$$

INPUT:

- `algorithm` – string (default: `"Konig"`); algorithm to use among:
  - `"Konig"` will compute a minimum vertex cover using Konig's algorithm (Wikipedia article Kőnig's_theorem_(graph_theory))
  - `"Cliquer"` will compute a minimum vertex cover using the Cliquer package
  - `"MILP"` will compute a minimum vertex cover through a mixed integer linear program
  - `"mcqd"` will use the MCQD solver (http://www.sicmm.org/~konc/maxclique/), and the MCQD package must be installed

- `value_only` – boolean (default: `False`); if set to `True`, only the size of a minimum vertex cover is returned. Otherwise, a minimum vertex cover is returned as a list of vertices.
- `reduction_rules` – (default: `True`); specify if the reductions rules from kernelization must be applied as pre-processing or not. See [ACFLSS04] for more details. Note that depending on the instance, it might be faster to disable reduction rules. This parameter is currently ignored when `algorithm == "Konig"`.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `sage.numerical.mip.MixedIntegerLinearProgram.solve()` of the class `sage.numerical.mip.MixedIntegerLinearProgram`.
- `verbosity` – non-negative integer (default: `0`); set the level of verbosity you want from the linear program solver. Since the problem of computing a vertex cover is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver. This option is only useful if `algorithm="MILP"`.

EXAMPLES:

On the Cycle Graph:

```
sage: B = BipartiteGraph(graphs.CycleGraph(6))
sage: len(B.vertex_cover())
3
sage: B.vertex_cover(value_only=True)
3
```

The two algorithms should return the same result:

```
sage: g = BipartiteGraph(graphs.RandomBipartite(10, 10, .5))
sage: vc1 = g.vertex_cover(algorithm="Konig")
sage: vc2 = g.vertex_cover(algorithm="Cliquer")
sage: len(vc1) == len(vc2)
True
```

1.5 View classes

This module implements views for (di)graphs. A view is a read-only iterable container enabling operations like `for e in E` and `e in E`. It is updated as the graph is updated. Hence, the graph should not be updated while iterating through a view. Views can be iterated multiple times.

Todo:

- View of neighborhood to get open/close neighborhood of a vertex/set of vertices, and also the vertex boundary

---

1.5.1 Classes

class sage.graphs.views.**EdgesView**

Bases: `object`

EdgesView class.

This class implements a read-only iterable container of edges enabling operations like `for e in E` and `e in E`. An *EdgesView* can be iterated multiple times, and checking membership is done in constant time. It avoids

the construction of edge lists and so consumes little memory. It is updated as the graph is updated. Hence, the graph should not be updated while iterating through an *EdgesView*.

INPUT:

- G – a (di)graph
- vertices – list (default: None); an iterable container of vertices or None. When set, consider only edges incident to specified vertices.
- labels – boolean (default: True); if False, each edge is simply a pair (u, v) of vertices
- ignore_direction – boolean (default: False); only applies to directed graphs. If True, searches across edges in either direction.
- sort – boolean (default: None); whether to sort edges
  - if None, sort edges according to the default ordering and give a deprecation warning as sorting will be set to False by default in the future
  - if True, edges are sorted according the ordering specified with parameter key
  - if False, edges are not sorted. This is the fastest and less memory consuming method for iterating over edges. This will become the default behavior in the future.
- key – a function (default: None); a function that takes an edge (a pair or a triple, according to the labels keyword) as its one argument and returns a value that can be used for comparisons in the sorting algorithm. This parameter is ignored when sort = False.

Warning: Since any object may be a vertex, there is no guarantee that any two vertices will be comparable, and thus no guarantee how two edges may compare. With default objects for vertices (all integers), or when all the vertices are of the same simple type, then there should not be a problem with how the vertices will be sorted. However, if you need to guarantee a total order for the sorting of the edges, use the key argument, as illustrated in the examples below.

EXAMPLES:

```
sage: from sage.graphs.views import EdgesView
sage: G = Graph([(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')])
sage: E = EdgesView(G, sort=True); E
[(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')]
sage: (1, 2) in E
False
sage: (1, 2, 'B') in E
True
sage: E = EdgesView(G, labels=False, sort=True); E
[(0, 1), (0, 2), (1, 2)]
sage: (1, 2) in E
True
sage: (1, 2, 'B') in E
False
sage: [e for e in E]
[(0, 1), (0, 2), (1, 2)]
```

An *EdgesView* can be iterated multiple times:

```
sage: G = graphs.CycleGraph(3)
sage: print(E)
[(0, 1), (0, 2), (1, 2)]
```

(continues on next page)

(continued from previous page)

```
sage: print(E)
[(0, 1), (0, 2), (1, 2)]
sage: for e in E:
....:     for ee in E:
....:         print((e, ee))
((0, 1), (0, 1))
((0, 1), (0, 2))
((0, 1), (1, 2))
((0, 2), (0, 1))
((0, 2), (0, 2))
((0, 2), (1, 2))
((1, 2), (0, 1))
((1, 2), (0, 2))
((1, 2), (1, 2))
```

We can check if a view is empty:

```
sage: E = EdgesView(graphs.CycleGraph(3), sort=False)
sage: if E:
....:     print('not empty')
not empty
sage: E = EdgesView(Graph(), sort=False)
sage: if not E:
....:     print('empty')
empty
```

When `sort` is `True`, edges are sorted by default in the default fashion:

```
sage: G = Graph([(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')])
sage: E = EdgesView(G, sort=True); E
[(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')]
```

This can be overridden by specifying a key function. This first example just ignores the labels in the third component of the triple:

```
sage: G = Graph([(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')])
sage: E = EdgesView(G, sort=True, key=lambda x: (x[1], -x[0])); E
[(0, 1, 'C'), (1, 2, 'B'), (0, 2, 'A')]
```

We can also sort according to the labels:

```
sage: G = Graph([(0, 1, 'C'), (0, 2, 'A'), (1, 2, 'B')])
sage: E = EdgesView(G, sort=True, key=lambda x: x[2]); E
[(0, 2, 'A'), (1, 2, 'B'), (0, 1, 'C')]
```

With a directed graph:

```
sage: G = digraphs.DeBruijn(2, 2)
sage: E = EdgesView(G, labels=False, sort=True); E
[('00', '00'), ('00', '01'), ('01', '10'), ('01', '11'),
 ('10', '00'), ('10', '01'), ('11', '10'), ('11', '11')]
sage: E = EdgesView(G, labels=False, sort=True, key=lambda e:(e[1], e[0])); E
[('00', '00'), ('10', '00'), ('00', '01'), ('10', '01'),
 ('01', '10'), ('11', '10'), ('01', '11'), ('11', '11')]
```

We can consider only edges incident to a specified set of vertices:

```
sage: G = graphs.CycleGraph(5)
sage: E = EdgesView(G, vertices=[0, 1], labels=False, sort=True); E
[(0, 1), (0, 4), (1, 2)]
sage: E = EdgesView(G, vertices=[0], labels=False, sort=True); E
[(0, 1), (0, 4)]
sage: E = EdgesView(G, vertices=None, labels=False, sort=True); E
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: G = digraphs.Circuit(5)
sage: E = EdgesView(G, vertices=[0, 1], labels=False, sort=True); E
[(0, 1), (1, 2)]
```

We can ignore the direction of the edges of a directed graph, in which case we search accross edges in either direction:

```
sage: G = digraphs.Circuit(5)
sage: E = EdgesView(G, vertices=[0, 1], labels=False, sort=True, ignore_
↪direction=False); E
[(0, 1), (1, 2)]
sage: (1, 0) in E
False
sage: E = EdgesView(G, vertices=[0, 1], labels=False, sort=True, ignore_
↪direction=True); E
[(0, 1), (0, 1), (1, 2), (4, 0)]
sage: (1, 0) in E
True
sage: G.has_edge(1, 0)
False
```

A view is updated as the graph is updated:

```
sage: G = Graph()
sage: E = EdgesView(G, vertices=[0, 3], labels=False, sort=True); E
[]
sage: G.add_edges([(0, 1), (1, 2)])
sage: E
[(0, 1)]
sage: G.add_edge(2, 3)
sage: E
[(0, 1), (2, 3)]
```

Hence, the graph should not be updated while iterating through a view:

```
sage: G = Graph([('a', 'b'), ('b', 'c')])
sage: E = EdgesView(G, labels=False, sort=False); E
[('a', 'b'), ('b', 'c')]
sage: for u, v in E:
....:     G.add_edge(u + u, v + v)
Traceback (most recent call last):
...
RuntimeError: dictionary changed size during iteration
```

Two *EdgesView* are considered equal if they report either both directed, or both undirected edges, they have the same settings for `ignore_direction`, they have the same settings for `labels`, and they report the same edges in the same order:

```
sage: G = graphs.HouseGraph()
sage: EG = EdgesView(G, sort=False)
sage: H = Graph(EG)
sage: EH = EdgesView(H, sort=False)
sage: EG == EH
True
sage: G.add_edge(0, 10)
sage: EG = EdgesView(G, sort=False)
sage: EG == EH
False
sage: H.add_edge(0, 10)
sage: EH = EdgesView(H, sort=False)
sage: EG == EH
True
sage: H = G.strong_orientation()
sage: EH = EdgesView(H, sort=False)
sage: EG == EH
False
```

The sum of two *EdgesView* is a list containing the edges in both *EdgesView*:

```
sage: E1 = EdgesView(Graph([(0, 1)]), labels=False, sort=False)
sage: E2 = EdgesView(Graph([(2, 3)]), labels=False, sort=False)
sage: E1 + E2
[(0, 1), (2, 3)]
sage: E2 + E1
[(2, 3), (0, 1)]
```

Recall that a *EdgesView* is read-only and that this method returns a list:

```
sage: E1 += E2
sage: type(E1) is list
True
```

It is also possible to get the sum a *EdgesView* with itself n times:

```
sage: E = EdgesView(Graph([(0, 1), (2, 3)]), labels=False, sort=True)
sage: E * 3
[(0, 1), (2, 3), (0, 1), (2, 3), (0, 1), (2, 3)]
sage: 3 * E
[(0, 1), (2, 3), (0, 1), (2, 3), (0, 1), (2, 3)]
```

Recall that a *EdgesView* is read-only and that this method returns a list:

```
sage: E *= 2
sage: type(E) is list
True
```

We can ask for the i-th edge, or a slice of the edges as a list:

```
sage: E = EdgesView(graphs.HouseGraph(), labels=False, sort=True)
sage: E[0]
(0, 1)
sage: E[2]
(1, 3)
sage: E[-1]
(3, 4)
```

(continues on next page)

(continued from previous page)

```
sage: E[1:-1]
[(0, 2), (1, 3), (2, 3), (2, 4)]
sage: E[::-1]
[(3, 4), (2, 4), (2, 3), (1, 3), (0, 2), (0, 1)]
```

CHAPTER

TWO

CONSTRUCTORS AND DATABASES

2.1 Common Graphs

All graphs in Sage can be built through the `graphs` object. In order to build a complete graph on 15 elements, one can do:

```
sage: g = graphs.CompleteGraph(15)
```

To get a path with 4 vertices, and the house graph:

```
sage: p = graphs.PathGraph(4)
sage: h = graphs.HouseGraph()
```

More interestingly, one can get the list of all graphs that Sage knows how to build by typing `graphs.` in Sage and then hitting tab.

Basic structures

AztecDiamondGraph	*CompleteMultipartiteGraph*	*LadderGraph*
BullGraph	*DiamondGraph*	*LollipopGraph*
ButterflyGraph	*DipoleGraph*	*PathGraph*
CircularLadderGraph	*EmptyGraph*	*StarGraph*
ClawGraph	*Grid2dGraph*	*TadpoleGraph*
CycleGraph	*GridGraph*	*ToroidalGrid2dGraph*
CompleteBipartiteGraph	*HouseGraph*	*Toroidal6RegularGrid2dGraph*
CompleteGraph	*HouseXGraph*	

Small Graphs

A small graph is just a single graph and has no parameter influencing the number of edges or vertices.

Balaban10Cage	*GolombGraph*	*MeredithGraph*
Balaban11Cage	*GossetGraph*	*MoebiusKantorGraph*
BidiakisCube	*GrayGraph*	*MoserSpindle*
BiggsSmithGraph	*GrotzschGraph*	*NauruGraph*
BlanusaFirstSnarkGraph	*HallJankoGraph*	*PappusGraph*
BlanusaSecondSnarkGraph	*HarborthGraph*	*PoussinGraph*
BrinkmannGraph	*HarriesGraph*	*PerkelGraph*
BrouwerHaemersGraph	*HarriesWongGraph*	*PetersenGraph*
BuckyBall	*HeawoodGraph*	*RobertsonGraph*
CameronGraph	*HerschelGraph*	*SchlaefliGraph*
Cell600	*HigmanSimsGraph*	*ShrikhandeGraph*
Cell120	*HoffmanGraph*	*SimsGewirtzGraph*
ChvatalGraph	*HoffmanSingletonGraph*	*SousselierGraph*
ClebschGraph	*HoltGraph*	*SylvesterGraph*
CoxeterGraph	*HortonGraph*	*SzekeresSnarkGraph*
DesarguesGraph	*IoninKharaghani765Graph*	*ThomsenGraph*
DejterGraph	*JankoKharaghaniGraph*	*TietzeGraph*
DoubleStarSnark	*JankoKharaghaniTonchevGraph*	*TruncatedIcosidodecahedralGraph*
DurerGraph	*KittellGraph*	*TruncatedTetrahedralGraph*
DyckGraph	*KrackhardtKiteGraph*	*Tutte12Cage*
EllinghamHorton54Graph	*Klein3RegularGraph*	*TutteCoxeterGraph*
EllinghamHorton78Graph	*Klein7RegularGraph*	*TutteGraph*
ErreraGraph	*LocalMcLaughlinGraph*	*U42Graph216*
F26AGraph	*LjubljanaGraph*	*U42Graph540*
FlowerSnark	*LivingstoneGraph*	*WagnerGraph*
FolkmanGraph	*M22Graph*	*WatkinsSnarkGraph*
FosterGraph	*MarkstroemGraph*	*WellsGraph*
FranklinGraph	*MathonStronglyRegularGraph*	*WienerArayaGraph*
FruchtGraph	*McGeeGraph*	*SuzukiGraph*
GoldnerHararyGraph	*McLaughlinGraph*	

Platonic solids (ordered ascending by number of vertices)

TetrahedralGraph	*HexahedralGraph*	*DodecahedralGraph*
OctahedralGraph	*IcosahedralGraph*	

Families of graphs

A family of graph is an infinite set of graphs which can be indexed by fixed number of parameters, e.g. two integer parameters. (A method whose name starts with a small letter does not return a single graph object but a graph iterator or a list of graphs or . . .)

BalancedTree	FuzzyBallGraph	NStarGraph
BarbellGraph	GeneralizedPetersenGraph	OddGraph
BubbleSortGraph	GoethalsSeidelGraph	PaleyGraph
CaiFurerImmermanGraph	HammingGraph	PasechnikGraph
chang_graphs	HanoiTowerGraph	petersen_family
CirculantGraph	HararyGraph	planar_graphs
cospectral_graphs	HyperStarGraph	quadrangulations
CubeGraph	JohnsonGraph	RingedTree
CubeConnectedCycle	KneserGraph	SierpinskiGasketGraph
DorogovtsevGoltsevMendesGraph	LCFGraph	SquaredSkewHadamardMatrixGraph
EgawaGraph	line_graph_forbidden_subgraphs	SwitchedSquaredSkewHadamardMatrixGrap
FibonacciTree	MathonPseudocyclicMergingGraph	strongly_regular_graph
FoldedCubeGraph	MathonPseudocyclicStronglyRegularGraph	SylvesterGraph
FriendshipGraph	MuzychukS6Graph	triangulations
fullerenes	MycielskiGraph	TuranGraph
FurerGadget	MycielskiStep	WheelGraph
fusenes	NKStarGraph	WindmillGraph

Graphs from classical geometries over finite fields

A number of classes of graphs related to geometries over finite fields and quadrics and Hermitean varieties there.

AffineOrthogonalPolarGraph	SymplecticDualPolarGraph	Nowhere0WordsTwoWeightCodeGraph
AhrensSzekeresGeneralizedQuadrangleGraph	SymplecticPolarGraph	HaemersGraph
NonisotropicOrthogonalPolarGraph	TaylorTwographDescendantSRG	OrthogonalDualPolarGraph
NonisotropicUnitaryPolarGraph	TaylorTwographSRG	UnitaryDualPolarGraph
OrthogonalPolarGraph	T2starGeneralizedQuadrangleGraph	UnitaryPolarGraph

Chessboard Graphs

BishopGraph	KnightGraph	RookGraph
KingGraph	QueenGraph	

Intersection graphs

These graphs are generated by geometric representations. The objects of the representation correspond to the graph vertices and the intersections of objects yield the graph edges.

IntersectionGraph	OrthogonalArrayBlockGraph	ToleranceGraph
IntervalGraph	PermutationGraph	

Random graphs

RandomBarabasiAlbert	RandomGNP	RandomShell
RandomBicubicPlanar	RandomHolmeKim	RandomToleranceGraph
RandomBipartite	RandomChordalGraph	RandomTree
RandomRegularBipartite	RandomIntervalGraph	RandomTreePowerlaw
RandomBlockGraph	RandomLobster	RandomTriangulation
RandomBoundedToleranceGraph	RandomNewmanWattsStrogatz	
RandomGNM	RandomRegular	

Graphs with a given degree sequence

DegreeSequence	*DegreeSequenceConfiguration*	*DegreeSequenceTree*
DegreeSequenceBipartite	*DegreeSequenceExpected*	

Miscellaneous

WorldMap	*AfricaMap*	
EuropeMap	*USAMap*	

AUTHORS:

- Robert Miller (2006-11-05): initial version, empty, random, petersen
- Emily Kirkman (2006-11-12): basic structures, node positioning for all constructors
- Emily Kirkman (2006-11-19): docstrings, examples
- William Stein (2006-12-05): Editing.
- Robert Miller (2007-01-16): Cube generation and plotting
- Emily Kirkman (2007-01-16): more basic structures, docstrings
- Emily Kirkman (2007-02-14): added more named graphs
- Robert Miller (2007-06-08-11): Platonic solids, random graphs, graphs with a given degree sequence, random directed graphs
- Robert Miller (2007-10-24): Isomorph free exhaustive generation
- Nathann Cohen (2009-08-12): WorldMap
- Michael Yurko (2009-9-01): added hyperstar, (n,k)-star, n-star, and bubblesort graphs
- Anders Jonsson (2009-10-15): added generalized Petersen graphs
- Harald Schilly and Yann Laigle-Chapuy (2010-03-24): added Fibonacci Tree
- Jason Grout (2010-06-04): cospectral_graphs
- Edward Scheinerman (2010-08-11): RandomTree
- Ed Scheinerman (2010-08-21): added Grotzsch graph and Mycielski graphs
- Ed Scheinerman (2010-11-15): added RandomTriangulation
- Minh Van Nguyen (2010-11-26): added more named graphs
- Keshav Kini (2011-02-16): added Shrikhande and Dyck graphs
- David Coudert (2012-02-10): new RandomGNP generator
- David Coudert (2012-08-02): added chessboard graphs: Queen, King, Knight, Bishop, and Rook graphs
- Nico Van Cleemput (2013-05-26): added fullerenes
- Nico Van Cleemput (2013-07-01): added benzenoids
- Birk Eisermann (2013-07-29): new section 'intersection graphs', added (random, bounded) tolerance graphs
- Marco Cognetta (2016-03-03): added TuranGraph

2.1.1 Functions and methods

class sage.graphs.graph_generators.**GraphGenerators**
Bases: object

A class consisting of constructors for several common graphs, as well as orderly generation of isomorphism class representatives. See the *module's help* for a list of supported constructors.

A list of all graphs and graph structures (other than isomorphism class representatives) in this database is available via tab completion. Type "graphs." and then hit the tab key to see which graphs are available.

The docstrings include educational information about each named graph with the hopes that this class can be used as a reference.

For all the constructors in this class (except the octahedral, dodecahedral, random and empty graphs), the position dictionary is filled to override the spring-layout algorithm.

ORDERLY GENERATION:

```
graphs(vertices, property=lambda x: True, augment='edges', size=None)
```

This syntax accesses the generator of isomorphism class representatives. Iterates over distinct, exhaustive representatives.

Also: see the use of the nauty package for generating graphs at the *nauty_geng()* method.

INPUT:

- vertices – a natural number or None to infinitely generate bigger and bigger graphs.
- property – (default: lambda x: True) any property to be tested on graphs before generation, but note that in general the graphs produced are not the same as those produced by using the property function to filter a list of graphs produced by using the lambda x: True default. The generation process assumes the property has certain characteristics set by the augment argument, and only in the case of inherited properties such that all subgraphs of the relevant kind (for augment='edges' or augment='vertices') of a graph with the property also possess the property will there be no missing graphs. (The property argument is ignored if degree_sequence is specified.)
- augment – (default: 'edges') possible values:
  - 'edges' – augments a fixed number of vertices by adding one edge. In this case, all graphs on *exactly* n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained from G by deleting one edge but not the vertices incident to that edge, satisfies the property, then this will generate all graphs with that property. If this does not hold, then all the graphs generated will satisfy the property, but there will be some missing.
  - 'vertices' – augments by adding a vertex and edges incident to that vertex. In this case, all graphs *up to* n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained from G by deleting one vertex and only edges incident to that vertex, satisfies the property, then this will generate all graphs with that property. If this does not hold, then all the graphs generated will satisfy the property, but there will be some missing.
- size – (default: None) the size of the graph to be generated.
- degree_sequence – (default: None) a sequence of non-negative integers, or None. If specified, the generated graphs will have these integers for degrees. In this case, property and size are both ignored.
- loops – (default: False) whether to allow loops in the graph or not.
- sparse – (default: True); whether to use a sparse or dense data structure. See the documentation of *Graph*.

- `copy` (boolean) – If set to `True` (default) this method makes copies of the graphs before returning them. If set to `False` the method returns the graph it is working on. The second alternative is faster, but modifying any of the graph instances returned by the method may break the function's behaviour, as it is using these graphs to compute the next ones: only use `copy = False` when you stick to *reading* the graphs returned.

EXAMPLES:

Print graphs on 3 or less vertices:

```
sage: for G in graphs(3, augment='vertices'):
....:     print(G)
Graph on 0 vertices
Graph on 1 vertex
Graph on 2 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 2 vertices
Graph on 3 vertices
```

Print graphs on 3 vertices.

```
sage: for G in graphs(3):
....:     print(G)
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
```

Generate all graphs with 5 vertices and 4 edges.

```
sage: L = graphs(5, size=4)
sage: len(list(L))
6
```

Generate all graphs with 5 vertices and up to 4 edges.

```
sage: L = list(graphs(5, lambda G: G.size() <= 4))
sage: len(L)
14
sage: graphs_list.show_graphs(L) # long time
```

Generate all graphs with up to 5 vertices and up to 4 edges.

```
sage: L = list(graphs(5, lambda G: G.size() <= 4, augment='vertices'))
sage: len(L)
31
sage: graphs_list.show_graphs(L)               # long time
```

Generate all graphs with degree at most 2, up to 6 vertices.

```
sage: property = lambda G: ( max([G.degree(v) for v in G] + [0]) <= 2 )
sage: L = list(graphs(6, property, augment='vertices'))
sage: len(L)
45
```

Generate all bipartite graphs on up to 7 vertices: (see OEIS sequence A033995)

```
sage: L = list( graphs(7, lambda G: G.is_bipartite(), augment='vertices') )
sage: [len([g for g in L if g.order() == i]) for i in [1..7]]
[1, 2, 3, 7, 13, 35, 88]
```

Generate all bipartite graphs on exactly 7 vertices:

```
sage: L = list( graphs(7, lambda G: G.is_bipartite()) )
sage: len(L)
88
```

Generate all bipartite graphs on exactly 8 vertices:

```
sage: L = list( graphs(8, lambda G: G.is_bipartite()) ) # long time
sage: len(L)                                            # long time
303
```

Remember that the property argument does not behave as a filter, except for appropriately inheritable properties:

```
sage: property = lambda G: G.is_vertex_transitive()
sage: len(list(graphs(4, property)))
1
sage: sum(1 for g in graphs(4) if property(g))
4

sage: property = lambda G: G.is_bipartite()
sage: len(list(graphs(4, property)))
7
sage: sum(1 for g in graphs(4) if property(g))
7
```

Generate graphs on the fly: (see OEIS sequence A000088)

```
sage: for i in range(7):
....:     print(len(list(graphs(i))))
1
1
2
4
11
34
156
```

Generate all simple graphs, allowing loops: (see OEIS sequence A000666)

```
sage: L = list(graphs(5,augment='vertices',loops=True))               # long time
sage: for i in [0..5]:  # long time
....:     print((i, len([g for g in L if g.order() == i]))) # long time
(0, 1)
(1, 2)
(2, 6)
(3, 20)
(4, 90)
(5, 544)
```

Generate all graphs with a specified degree sequence (see OEIS sequence A002851):

```
sage: for i in [4,6,8]:  # long time (4s on sage.math, 2012)
....:     print((i, len([g for g in graphs(i, degree_sequence=[3]*i) if g.is_
↪connected()])))
(4, 1)
(6, 2)
(8, 5)
sage: for i in [4,6,8]:  # long time (7s on sage.math, 2012)
....:     print((i, len([g for g in graphs(i, augment='vertices', degree_
↪sequence=[3]*i) if g.is_connected()])))
(4, 1)
(6, 2)
(8, 5)
```

```
sage: print((10, len([g for g in graphs(10,degree_sequence=[3]*10) if g.is_
↪connected()]))) # not tested
(10, 19)
```

Make sure that the graphs are really independent and the generator survives repeated vertex removal (trac ticket #8458):

```
sage: for G in graphs(3):
....:     G.delete_vertex(0)
....:     print(G.order())
2
2
2
2
```

REFERENCE:

- Brendan D. McKay, Isomorph-Free Exhaustive generation. *Journal of Algorithms*, Volume 26, Issue 2, February 1998, pages 306-324.

static AffineOrthogonalPolarGraph(d, q, *sign='+'*)

Returns the affine polar graph $VO^+(d,q), VO^-(d,q)$ or $VO(d,q)$.

Affine Polar graphs are built from a d-dimensional vector space over F_q, and a quadratic form which is hyperbolic, elliptic or parabolic according to the value of `sign`.

Note that $VO^+(d,q), VO^-(d,q)$ are strongly regular graphs, while $VO(d,q)$ is not.

For more information on Affine Polar graphs, see Affine Polar Graphs page of Andries Brouwer's website.

INPUT:

- `d` (integer) – `d` must be even if `sign is not None`, and odd otherwise.
- `q` (integer) – a power of a prime number, as F_q must exist.
- `sign` – must be equal to `"+"`, `"-"`, or `None` to compute (respectively) $VO^+(d,q), VO^-(d,q)$ or $VO(d,q)$. By default `sign="+"`.

Note: The graph $VO^\epsilon(d,q)$ is the graph induced by the non-neighbors of a vertex in an *Orthogonal Polar Graph* $O^\epsilon(d+2,q)$.

EXAMPLES:

The *Brouwer-Haemers graph* is isomorphic to $VO^-(4,3)$:

```
sage: g = graphs.AffineOrthogonalPolarGraph(4,3,"-")
sage: g.is_isomorphic(graphs.BrouwerHaemersGraph())
True
```

Some examples from Brouwer's table or strongly regular graphs:

```
sage: g = graphs.AffineOrthogonalPolarGraph(6,2,"-"); g
Affine Polar Graph VO^-(6,2): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 27, 10, 12)
sage: g = graphs.AffineOrthogonalPolarGraph(6,2,"+"); g
Affine Polar Graph VO^+(6,2): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 35, 18, 20)
```

When `sign is None`:

```
sage: g = graphs.AffineOrthogonalPolarGraph(5,2,None); g
Affine Polar Graph VO^-(5,2): Graph on 32 vertices
sage: g.is_strongly_regular(parameters=True)
False
sage: g.is_regular()
True
sage: g.is_vertex_transitive()
True
```

static AfricaMap(*continental=False*, *year=2018*)

Return African states as a graph of common border.

"African state" here is defined as an independent state having the capital city in Africa. The graph has an edge between those countries that have common *land* border.

INPUT:

- `continental`, a Boolean – if set, only return states in the continental Africa
- `year` – reserved for future use

EXAMPLES:

```
sage: Africa = graphs.AfricaMap(); Africa
Africa Map: Graph on 54 vertices
sage: sorted(Africa.neighbors('Libya'))
['Algeria', 'Chad', 'Egypt', 'Niger', 'Sudan', 'Tunisia']

sage: cont_Africa = graphs.AfricaMap(continental=True)
sage: cont_Africa.order()
48
sage: 'Madagaskar' in cont_Africa
False
```

static AhrensSzekeresGeneralizedQuadrangleGraph(q, *dual=False*)

Return the collinearity graph of the generalized quadrangle $AS(q)$, or of its dual

Let q be an odd prime power. $AS(q)$ is a generalized quadrangle (Wikipedia article Generalized_quadrangle) of order $(q-1, q+1)$, see 3.1.5 in [PT2009]. Its points are elements of F_q^3, and lines are sets of size q of the form

- $\{(\sigma, a, b) \mid \sigma \in F_q\}$

- $\{(a, \sigma, b) \mid \sigma \in F_q\}$
- $\{(c\sigma^2 - b\sigma + a, -2c\sigma + b, \sigma) \mid \sigma \in F_q\}$,

where a, b, c are arbitrary elements of F_q.

INPUT:

- `q` – a power of an odd prime number
- `dual` – if `False` (default), return the collinearity graph of $AS(q)$. Otherwise return the collinearity graph of the dual $AS(q)$.

EXAMPLES:

```
sage: g=graphs.AhrensSzekeresGeneralizedQuadrangleGraph(5); g
AS(5); GQ(4, 6): Graph on 125 vertices
sage: g.is_strongly_regular(parameters=True)
(125, 28, 3, 7)
sage: g=graphs.AhrensSzekeresGeneralizedQuadrangleGraph(5,dual=True); g
AS(5)*; GQ(6, 4): Graph on 175 vertices
sage: g.is_strongly_regular(parameters=True)
(175, 30, 5, 5)
```

static AztecDiamondGraph(*n*)

Return the Aztec Diamond graph of order `n`.

See the Wikipedia article Aztec_diamond for more information.

EXAMPLES:

```
sage: graphs.AztecDiamondGraph(2)
Aztec Diamond graph of order 2

sage: [graphs.AztecDiamondGraph(i).num_verts() for i in range(8)]
[0, 4, 12, 24, 40, 60, 84, 112]

sage: [graphs.AztecDiamondGraph(i).num_edges() for i in range(8)]
[0, 4, 16, 36, 64, 100, 144, 196]

sage: G = graphs.AztecDiamondGraph(3)
sage: sum(1 for p in G.perfect_matchings())
64
```

static Balaban10Cage(*embedding=1*)

Return the Balaban 10-cage.

The Balaban 10-cage is a 3-regular graph with 70 vertices and 105 edges. See the Wikipedia article Balaban_10-cage.

The default embedding gives a deeper understanding of the graph's automorphism group. It is divided into 4 layers (each layer being a set of points at equal distance from the drawing's center). From outside to inside:

- L1: The outer layer (vertices which are the furthest from the origin) is actually the disjoint union of two cycles of length 10.
- L2: The second layer is an independent set of 20 vertices.
- L3: The third layer is a matching on 10 vertices.
- L4: The inner layer (vertices which are the closest from the origin) is also the disjoint union of two cycles of length 10.

This graph is not vertex-transitive, and its vertices are partitioned into 3 orbits: L2, L3, and the union of L1 of L4 whose elements are equivalent.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to be either 1 or 2.

EXAMPLES:

```
sage: g = graphs.Balaban10Cage()
sage: g.girth()
10
sage: g.chromatic_number()
2
sage: g.diameter()
6
sage: g.is_hamiltonian()
True
sage: g.show(figsize=[10,10])    # long time
```

static Balaban11Cage(*embedding=1*)

Return the Balaban 11-cage.

For more information, see the Wikipedia article Balaban_11-cage.

INPUT:

- `embedding` – three embeddings are available, and can be selected by setting `embedding` to be 1, 2, or 3.
  - The first embedding is the one appearing on page 9 of the Fifth Annual Graph Drawing Contest report [EMMN1998]. It separates vertices based on their eccentricity (see *eccentricity()*).
  - The second embedding has been produced just for Sage and is meant to emphasize the automorphism group's 6 orbits.
  - The last embedding is the default one produced by the *LCFGraph()* constructor.

Note: The vertex labeling changes according to the value of `embedding=1`.

EXAMPLES:

Basic properties:

```
sage: g = graphs.Balaban11Cage()
sage: g.order()
112
sage: g.size()
168
sage: g.girth()
11
sage: g.diameter()
8
sage: g.automorphism_group().cardinality()
64
```

Our many embeddings:

```
sage: g1 = graphs.Balaban11Cage(embedding=1)
sage: g2 = graphs.Balaban11Cage(embedding=2)
sage: g3 = graphs.Balaban11Cage(embedding=3)
sage: g1.show(figsize=[10,10])    # long time
sage: g2.show(figsize=[10,10])    # long time
sage: g3.show(figsize=[10,10])    # long time
```

Proof that the embeddings are the same graph:

```
sage: g1.is_isomorphic(g2) # g2 and g3 are obviously isomorphic
True
```

static BalancedTree(*r*, *h*)

Returns the perfectly balanced tree of height $h \geq 1$, whose root has degree $r \geq 2$.

The number of vertices of this graph is $1 + r + r^2 + \cdots + r^h$, that is, $\frac{r^{h+1}-1}{r-1}$. The number of edges is one less than the number of vertices.

INPUT:

- `r` – positive integer ≥ 2. The degree of the root node.
- `h` – positive integer ≥ 1. The height of the balanced tree.

OUTPUT:

The perfectly balanced tree of height $h \geq 1$ and whose root has degree $r \geq 2$. A `NetworkXError` is returned if $r < 2$ or $h < 1$.

ALGORITHM:

Uses NetworkX.

EXAMPLES:

A balanced tree whose root node has degree $r = 2$, and of height $h = 1$, has order 3 and size 2:

```
sage: G = graphs.BalancedTree(2, 1); G
Balanced tree: Graph on 3 vertices
sage: G.order(); G.size()
3
2
sage: r = 2; h = 1
sage: v = 1 + r
sage: v; v - 1
3
2
```

Plot a balanced tree of height 5, whose root node has degree $r = 3$:

```
sage: G = graphs.BalancedTree(3, 5)
sage: G.show()    # long time
```

A tree is bipartite. If its vertex set is finite, then it is planar.

```
sage: r = randint(2, 5); h = randint(1, 7)
sage: T = graphs.BalancedTree(r, h)
sage: T.is_bipartite()
True
sage: T.is_planar()
```

(continues on next page)

(continued from previous page)

```
True
sage: v = (r^(h + 1) - 1) / (r - 1)
sage: T.order() == v
True
sage: T.size() == v - 1
True
```

static BarbellGraph(*n1*, *n2*)

Returns a barbell graph with `2*n1 + n2` nodes. The argument `n1` must be greater than or equal to 2.

A barbell graph is a basic structure that consists of a path graph of order `n2` connecting two complete graphs of order `n1` each.

INPUT:

- `n1` – integer ≥ 2. The order of each of the two complete graphs.
- `n2` – nonnegative integer. The order of the path graph connecting the two complete graphs.

OUTPUT:

A barbell graph of order `2*n1 + n2`. A `ValueError` is returned if `n1 < 2` or `n2 < 0`.

PLOTTING:

Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each barbell graph will be displayed with the two complete graphs in the lower-left and upper-right corners, with the path graph connecting diagonally between the two. Thus the `n1`-th node will be drawn at a 45 degree angle from the horizontal right center of the first complete graph, and the `n1 + n2 + 1`-th node will be drawn 45 degrees below the left horizontal center of the second complete graph.

EXAMPLES:

Construct and show a barbell graph `Bar = 4, Bells = 9`:

```
sage: g = graphs.BarbellGraph(9, 4); g
Barbell graph: Graph on 22 vertices
sage: g.show() # long time
```

An `n1 >= 2, n2 >= 0` barbell graph has order `2*n1 + n2`. It has the complete graph on `n1` vertices as a subgraph. It also has the path graph on `n2` vertices as a subgraph.

```
sage: n1 = randint(2, 2*10^2)
sage: n2 = randint(0, 2*10^2)
sage: g = graphs.BarbellGraph(n1, n2)
sage: v = 2*n1 + n2
sage: g.order() == v
True
sage: K_n1 = graphs.CompleteGraph(n1)
sage: P_n2 = graphs.PathGraph(n2)
sage: s_K = g.subgraph_search(K_n1, induced=True)
sage: s_P = g.subgraph_search(P_n2, induced=True)
sage: K_n1.is_isomorphic(s_K)
True
sage: P_n2.is_isomorphic(s_P)
True
```

static BidiakisCube()

Return the Bidiakis cube.

For more information, see the Wikipedia article Bidiakis_cube.

EXAMPLES:

The Bidiakis cube is a 3-regular graph having 12 vertices and 18 edges. This means that each vertex has a degree of 3.

```
sage: g = graphs.BidiakisCube(); g
Bidiakis cube: Graph on 12 vertices
sage: g.show()  # long time
sage: g.order()
12
sage: g.size()
18
sage: g.is_regular(3)
True
```

It is a Hamiltonian graph with diameter 3 and girth 4:

```
sage: g.is_hamiltonian()
True
sage: g.diameter()
3
sage: g.girth()
4
```

It is a planar graph with characteristic polynomial $(x-3)(x-2)(x^4)(x+1)(x+2)(x^2+x-4)^2$ and chromatic number 3:

```
sage: g.is_planar()
True
sage: bool(g.characteristic_polynomial() == expand((x - 3) * (x - 2) * (x^4)␣
↪* (x + 1) * (x + 2) * (x^2 + x - 4)^2))
True
sage: g.chromatic_number()
3
```

static BiggsSmithGraph(*embedding=1*)

Return the Biggs-Smith graph.

For more information, see the Wikipedia article Biggs-Smith_graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to be 1 or 2.

EXAMPLES:

Basic properties:

```
sage: g = graphs.BiggsSmithGraph()
sage: g.order()
102
sage: g.size()
153
sage: g.girth()
9
sage: g.diameter()
7
sage: g.automorphism_group().cardinality() # long time
```

(continues on next page)

(continued from previous page)

```
2448
sage: g.show(figsize=[10, 10])    # long time
```

The other embedding:

```
sage: graphs.BiggsSmithGraph(embedding=2).show() # long time
```

static BishopGraph(*dim_list*, *radius=None*, *relabel=False*)

Returns the d-dimensional Bishop Graph with prescribed dimensions.

The 2-dimensional Bishop Graph of parameters n and m is a graph with nm vertices in which each vertex represents a square in an $n \times m$ chessboard, and each edge corresponds to a legal move by a bishop.

The d-dimensional Bishop Graph with $d >= 2$ has for vertex set the cells of a d-dimensional grid with prescribed dimensions, and each edge corresponds to a legal move by a bishop in any pairs of dimensions.

The Bishop Graph is not connected.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `radius` – (default: `None`) by setting the radius to a positive integer, one may decrease the power of the bishop to at most `radius` steps.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

EXAMPLES:

The (n,m)-Bishop Graph is not connected:

```
sage: G = graphs.BishopGraph( [3, 4] )
sage: G.is_connected()
False
```

The Bishop Graph can be obtained from Knight Graphs:

```
sage: for d in range(3,12):    # long time
....:     H = Graph()
....:     for r in range(1,d+1):
....:         B = graphs.BishopGraph([d,d],radius=r)
....:         H.add_edges( graphs.KnightGraph([d,d],one=r,two=r).edges() )
....:         if not B.is_isomorphic(H):
....:             print("that's not good!")
```

static BlanusaFirstSnarkGraph()

Return the first Blanusa Snark Graph.

The Blanusa graphs are two snarks on 18 vertices and 27 edges. For more information on them, see the Wikipedia article Blanusa_snarks.

See also:

- *BlanusaSecondSnarkGraph()*.

EXAMPLES:

```
sage: g = graphs.BlanusaFirstSnarkGraph()
sage: g.order()
18
sage: g.size()
27
sage: g.diameter()
4
sage: g.girth()
5
sage: g.automorphism_group().cardinality()
8
```

static BlanusaSecondSnarkGraph()

Return the second Blanusa Snark Graph.

The Blanusa graphs are two snarks on 18 vertices and 27 edges. For more information on them, see the Wikipedia article Blanusa_snarks.

See also:

- *BlanusaFirstSnarkGraph()*.

EXAMPLES:

```
sage: g = graphs.BlanusaSecondSnarkGraph()
sage: g.order()
18
sage: g.size()
27
sage: g.diameter()
4
sage: g.girth()
5
sage: g.automorphism_group().cardinality()
4
```

static BrinkmannGraph()

Return the Brinkmann graph.

For more information, see the Wikipedia article Brinkmann_graph.

EXAMPLES:

The Brinkmann graph is a 4-regular graph having 21 vertices and 42 edges. This means that each vertex has degree 4.

```
sage: G = graphs.BrinkmannGraph(); G
Brinkmann graph: Graph on 21 vertices
sage: G.show()  # long time
sage: G.order()
21
sage: G.size()
42
sage: G.is_regular(4)
True
```

It is an Eulerian graph with radius 3, diameter 3, and girth 5.

```
sage: G.is_eulerian()
True
sage: G.radius()
3
sage: G.diameter()
3
sage: G.girth()
5
```

The Brinkmann graph is also Hamiltonian with chromatic number 4:

```
sage: G.is_hamiltonian()
True
sage: G.chromatic_number()
4
```

Its automorphism group is isomorphic to D_7:

```
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(7))
True
```

static BrouwerHaemersGraph()

Return the Brouwer-Haemers Graph.

The Brouwer-Haemers is the only strongly regular graph of parameters $(81, 20, 1, 6)$. It is build in Sage as the Affine Orthogonal graph $VO^-(6, 3)$. For more information on this graph, see its corresponding page on Andries Brouwer's website.

EXAMPLES:

```
sage: g = graphs.BrouwerHaemersGraph()
sage: g
Brouwer-Haemers: Graph on 81 vertices
```

It is indeed strongly regular with parameters $(81, 20, 1, 6)$:

```
sage: g.is_strongly_regular(parameters = True) # long time
(81, 20, 1, 6)
```

Its has as eigenvalues $20, 2$ and -7:

```
sage: set(g.spectrum()) == {20,2,-7}
True
```

static BubbleSortGraph(n)

Returns the bubble sort graph $B(n)$.

The vertices of the bubble sort graph are the set of permutations on n symbols. Two vertices are adjacent if one can be obtained from the other by swapping the labels in the i-th and $(i + 1)$-th positions for $1 \leq i \leq n - 1$. In total, $B(n)$ has order $n!$. Swapping two labels as described previously corresponds to multiplying on the right the permutation corresponding to the node by an elementary transposition in the SymmetricGroup.

The bubble sort graph is the underlying graph of the permutahedron().

INPUT:

- n – positive integer. The number of symbols to permute.

OUTPUT:

The bubble sort graph $B(n)$ on n symbols. If $n < 1$, a `ValueError` is returned.

EXAMPLES:

```
sage: g = graphs.BubbleSortGraph(4); g
Bubble sort: Graph on 24 vertices
sage: g.plot() # long time
Graphics object consisting of 61 graphics primitives
```

The bubble sort graph on $n = 1$ symbol is the trivial graph K_1:

```
sage: graphs.BubbleSortGraph(1)
Bubble sort: Graph on 1 vertex
```

If $n \geq 1$, then the order of $B(n)$ is $n!$:

```
sage: n = randint(1, 8)
sage: g = graphs.BubbleSortGraph(n)
sage: g.order() == factorial(n)
True
```

See also:

- `permutahedron()`

AUTHORS:

- Michael Yurko (2009-09-01)

static BuckyBall()

Create the Bucky Ball graph.

This graph is a 3-regular 60-vertex planar graph. Its vertices and edges correspond precisely to the carbon atoms and bonds in buckminsterfullerene. When embedded on a sphere, its 12 pentagon and 20 hexagon faces are arranged exactly as the sections of a soccer ball.

EXAMPLES:

The Bucky Ball is planar.

```
sage: g = graphs.BuckyBall()
sage: g.is_planar()
True
```

The Bucky Ball can also be created by extracting the 1-skeleton of the Bucky Ball polyhedron, but this is much slower.

```
sage: g = polytopes.buckyball().vertex_graph()
sage: g.remove_loops()
sage: h = graphs.BuckyBall()
sage: g.is_isomorphic(h)
True
```

The graph is returned along with an attractive embedding.

```
sage: g = graphs.BuckyBall()
sage: g.plot(vertex_labels=False, vertex_size=10).show() # long time
```

static BullGraph()

Returns a bull graph with 5 nodes.

A bull graph is named for its shape. It's a triangle with horns. See the Wikipedia article Bull_graph for more information.

PLOTTING:

Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the bull graph is drawn as a triangle with the first node (0) on the bottom. The second and third nodes (1 and 2) complete the triangle. Node 3 is the horn connected to 1 and node 4 is the horn connected to node 2.

EXAMPLES:

Construct and show a bull graph:

```
sage: g = graphs.BullGraph(); g
Bull graph: Graph on 5 vertices
sage: g.show() # long time
```

The bull graph has 5 vertices and 5 edges. Its radius is 2, its diameter 3, and its girth 3. The bull graph is planar with chromatic number 3 and chromatic index also 3.

```
sage: g.order(); g.size()
5
5
sage: g.radius(); g.diameter(); g.girth()
2
3
3
sage: g.chromatic_number()
3
```

The bull graph has chromatic polynomial $x(x-2)(x-1)^3$ and Tutte polynomial $x^4+x^3+x^2y$. Its characteristic polynomial is $x(x^2-x-3)(x^2+x-1)$, which follows from the definition of characteristic polynomials for graphs, i.e. $\det(xI-A)$, where x is a variable, A the adjacency matrix of the graph, and I the identity matrix of the same dimensions as A.

```
sage: chrompoly = g.chromatic_polynomial()
sage: bool(expand(x * (x - 2) * (x - 1)^3) == chrompoly)
True
sage: charpoly = g.characteristic_polynomial()
sage: M = g.adjacency_matrix(); M
[0 1 1 0 0]
[1 0 1 1 0]
[1 1 0 0 1]
[0 1 0 0 0]
[0 0 1 0 0]
sage: Id = identity_matrix(ZZ, M.nrows())
sage: D = x*Id - M
sage: bool(D.determinant() == charpoly)
True
sage: bool(expand(x * (x^2 - x - 3) * (x^2 + x - 1)) == charpoly)
True
```

static ButterflyGraph()

Returns the butterfly graph.

Let C_3 be the cycle graph on 3 vertices. The butterfly or bowtie graph is obtained by joining two copies of C_3 at a common vertex, resulting in a graph that is isomorphic to the friendship graph F_2. See the Wikipedia article Butterfly_graph for more information.

See also:

- *GraphGenerators.FriendshipGraph()*

EXAMPLES:

The butterfly graph is a planar graph on 5 vertices and having 6 edges.

```
sage: G = graphs.ButterflyGraph(); G
Butterfly graph: Graph on 5 vertices
sage: G.show()  # long time
sage: G.is_planar()
True
sage: G.order()
5
sage: G.size()
6
```

It has diameter 2, girth 3, and radius 1.

```
sage: G.diameter()
2
sage: G.girth()
3
sage: G.radius()
1
```

The butterfly graph is Eulerian, with chromatic number 3.

```
sage: G.is_eulerian()
True
sage: G.chromatic_number()
3
```

static CaiFurerImmermanGraph(*G*, *twisted=False*)

Return the a Cai-Furer-Immerman graph from G, possibly a twisted one, and a partition of its nodes.

A Cai-Furer-Immerman graph from/on G is a graph created by applying the transformation described in [CFI1992] on a graph G, that is substituting every vertex v in G with a Furer gadget $F(v)$ of order d equal to the degree of the vertex, and then substituting every edge (v, u) in G with a pair of edges, one connecting the two "a" nodes of $F(v)$ and $F(u)$ and the other their two "b" nodes. The returned coloring of the vertices is made by the union of the colorings of each single Furer gadget, individualised for each vertex of G. To understand better what these "a" and "b" nodes are, see the documentation on Furer gadgets.

Furthermore, this method can apply what is described in the paper mentioned above as a "twist" on an edge, that is taking only one of the pairs of edges introduced in the new graph and swap two of their extremes, making each edge go from an "a" node to a "b" node. This is only doable if the original graph G is connected.

A CaiFurerImmerman graph on a graph with no balanced vertex separators smaller than s and its twisted version cannot be distinguished by k-WL for any k < s.

INPUT:

- **G – An undirected graph on which to construct the** Cai-Furer-Immerman graph

- **twisted – A boolean indicating if the version to construct** is a twisted one or not

OUTPUT:

- H – The Cai-Furer-Immerman graph on G
- **coloring – A list of list of vertices, representing the** partition induced by the coloring on H

EXAMPLES:

CaiFurerImmerman graph with no balanced vertex separator smaller than 2

```
sage: G = graphs.CycleGraph(4)
sage: CFI, p = graphs.CaiFurerImmermanGraph(G)
sage: sorted(CFI, key=str)
[(0, ()), (0, (0, 'a')), (0, (0, 'b')), (0, (0, 1)), (0, (1, 'a')),
 (0, (1, 'b')), (1, ()), (1, (0, 'a')), (1, (0, 'b')), (1, (0, 1)),
 (1, (1, 'a')), (1, (1, 'b')), (2, ()), (2, (0, 'a')), (2, (0, 'b')),
 (2, (0, 1)), (2, (1, 'a')), (2, (1, 'b')), (3, ()), (3, (0, 'a')),
 (3, (0, 'b')), (3, (0, 1)), (3, (1, 'a')), (3, (1, 'b'))]
sage: sorted(CFI.edge_iterator(), key=str)
[((0, ()), (0, (0, 'b')), None),
 ((0, ()), (0, (1, 'b')), None),
 ((0, (0, 'a')), (1, (0, 'a')), None),
 ((0, (0, 'b')), (1, (0, 'b')), None),
 ((0, (0, 1)), (0, (0, 'a')), None),
 ((0, (0, 1)), (0, (1, 'a')), None),
 ((0, (1, 'a')), (3, (0, 'a')), None),
 ((0, (1, 'b')), (3, (0, 'b')), None),
 ((1, ()), (1, (0, 'b')), None),
 ((1, ()), (1, (1, 'b')), None),
 ((1, (0, 1)), (1, (0, 'a')), None),
 ((1, (0, 1)), (1, (1, 'a')), None),
 ((1, (1, 'a')), (2, (0, 'a')), None),
 ((1, (1, 'b')), (2, (0, 'b')), None),
 ((2, ()), (2, (0, 'b')), None),
 ((2, ()), (2, (1, 'b')), None),
 ((2, (0, 1)), (2, (0, 'a')), None),
 ((2, (0, 1)), (2, (1, 'a')), None),
 ((2, (1, 'a')), (3, (1, 'a')), None),
 ((2, (1, 'b')), (3, (1, 'b')), None),
 ((3, ()), (3, (0, 'b')), None),
 ((3, ()), (3, (1, 'b')), None),
 ((3, (0, 1)), (3, (0, 'a')), None),
 ((3, (0, 1)), (3, (1, 'a')), None)]
```

static CameronGraph()

Return the Cameron graph.

The Cameron graph is strongly regular with parameters $v = 231, k = 30, \lambda = 9, \mu = 3$.

For more information on the Cameron graph, see https://www.win.tue.nl/~aeb/graphs/Cameron.html.

EXAMPLES:

```
sage: g = graphs.CameronGraph()
sage: g.order()
231
sage: g.size()
3465
```

(continues on next page)

(continued from previous page)

```
sage: g.is_strongly_regular(parameters = True) # long time
(231, 30, 9, 3)
```

static Cell120()

Return the 120-Cell graph.

This is the adjacency graph of the 120-cell. It has 600 vertices and 1200 edges. For more information, see the Wikipedia article 120-cell.

EXAMPLES:

```
sage: g = graphs.Cell120()       # long time
sage: g.size()                   # long time
1200
sage: g.is_regular(4)            # long time
True
sage: g.is_vertex_transitive()   # long time
True
```

static Cell600(*embedding=1*)

Return the 600-Cell graph.

This is the adjacency graph of the 600-cell. It has 120 vertices and 720 edges. For more information, see the Wikipedia article 600-cell.

INPUT:

- `embedding` (1 (default) or 2) – two different embeddings for a plot.

EXAMPLES:

```
sage: g = graphs.Cell600()       # long time
sage: g.size()                   # long time
720
sage: g.is_regular(12)           # long time
True
sage: g.is_vertex_transitive()   # long time
True
```

static ChessboardGraphGenerator(*dim_list*, *rook=True*, *rook_radius=None*, *bishop=True*, *bishop_radius=None*, *knight=True*, *knight_x=1*, *knight_y=2*, *relabel=False*)

Returns a Graph built on a d-dimensional chessboard with prescribed dimensions and interconnections.

This function allows to generate many kinds of graphs corresponding to legal movements on a d-dimensional chessboard: Queen Graph, King Graph, Knight Graphs, Bishop Graph, and many generalizations. It also allows to avoid redundant code.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `rook` – (default: `True`) boolean value indicating if the chess piece is able to move as a rook, that is at any distance along a dimension.
- `rook_radius` – (default: None) integer value restricting the rook-like movements to distance at most $rook_radius$.
- `bishop` – (default: `True`) boolean value indicating if the chess piece is able to move like a bishop, that is along diagonals.

- `bishop_radius` – (default: None) integer value restricting the bishop-like movements to distance at most $bishop_radius$.
- `knight` – (default: `True`) boolean value indicating if the chess piece is able to move like a knight.
- `knight_x` – (default: 1) integer indicating the number on steps the chess piece moves in one dimension when moving like a knight.
- `knight_y` – (default: 2) integer indicating the number on steps the chess piece moves in the second dimension when moving like a knight.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

OUTPUT:

- A Graph build on a d-dimensional chessboard with prescribed dimensions, and with edges according given parameters.
- A string encoding the dimensions. This is mainly useful for providing names to graphs.

EXAMPLES:

A $(2, 2)$-King Graph is isomorphic to the complete graph on 4 vertices:

```
sage: G, _ = graphs.ChessboardGraphGenerator( [2,2] )
sage: G.is_isomorphic( graphs.CompleteGraph(4) )
True
```

A Rook's Graph in 2 dimensions is isomorphic to the Cartesian product of 2 complete graphs:

```
sage: G, _ = graphs.ChessboardGraphGenerator( [3,4], rook=True, rook_
↪radius=None, bishop=False, knight=False )
sage: H = ( graphs.CompleteGraph(3) ).cartesian_product( graphs.
↪CompleteGraph(4) )
sage: G.is_isomorphic(H)
True
```

static ChvatalGraph()

Return the Chvatal graph.

Chvatal graph is one of the few known graphs to satisfy Grunbaum's conjecture that for every m, n, there is an m-regular, m-chromatic graph of girth at least n. For more information, see the Wikipedia article Chv%C3%A1tal_graph.

EXAMPLES:

The Chvatal graph has 12 vertices and 24 edges. It is a 4-regular, 4-chromatic graph with radius 2, diameter 2, and girth 4.

```
sage: G = graphs.ChvatalGraph(); G
Chvatal graph: Graph on 12 vertices
sage: G.order(); G.size()
12
24
sage: G.degree()
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
sage: G.chromatic_number()
4
sage: G.radius(); G.diameter(); G.girth()
2
2
4
```

static CirculantGraph(*n*, *adjacency*)

Returns a circulant graph with n nodes.

A circulant graph has the property that the vertex i is connected with the vertices $i + j$ and $i - j$ for each j in `adjacency`.

INPUT:

- `n` - number of vertices in the graph
- `adjacency` - the list of j values

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each circulant graph will be displayed with the first (0) node at the top, with the rest following in a counterclockwise manner.

Filling the position dictionary in advance adds O(n) to the constructor.

See also:

- *sage.graphs.generic_graph.GenericGraph.is_circulant()* – checks whether a (di)graph is circulant, and/or returns all possible sets of parameters.

EXAMPLES: Compare plotting using the predefined layout and networkx:

```
sage: import networkx
sage: n = networkx.cycle_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.CirculantGraph(23,2)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

We next view many cycle graphs as a Sage graphics array. First we use the `CirculantGraph` constructor, which fills in the position dictionary:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CirculantGraph(i+4, i+1)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

Compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.cycle_graph(i+3)
....:     k = Graph(spr)
....:     g.append(k)
sage: for i in range(3):
....:  n = []
....:  for m in range(3):
....:      n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
```

(continues on next page)

(continued from previous page)

```
....:  j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

Passing a 1 into adjacency should give the cycle.

```
sage: graphs.CirculantGraph(6,1)==graphs.CycleGraph(6)
True
sage: graphs.CirculantGraph(7,[1,3]).edges(labels=false)
[(0, 1),
(0, 3),
(0, 4),
(0, 6),
(1, 2),
(1, 4),
(1, 5),
(2, 3),
(2, 5),
(2, 6),
(3, 4),
(3, 6),
(4, 5),
(5, 6)]
```

static CircularLadderGraph(n)

Return a circular ladder graph with $2 * n$ nodes.

A Circular ladder graph is a ladder graph that is connected at the ends, i.e.: a ladder bent around so that top meets bottom. Thus it can be described as two parallel cycle graphs connected at each corresponding node pair.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the circular ladder graph is displayed as an inner and outer cycle pair, with the first n nodes drawn on the inner circle. The first (0) node is drawn at the top of the inner-circle, moving clockwise after that. The outer circle is drawn with the $(n+1)$, we rotate the outer circle by an angle of $\pi/8$ to ensure that all edges are visible (otherwise the 4 vertices of the graph would be placed on a single line).

EXAMPLES:

Construct and show a circular ladder graph with 26 nodes:

```
sage: g = graphs.CircularLadderGraph(13)
sage: g.show() # long time
```

Create several circular ladder graphs in a Sage graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CircularLadderGraph(i+3)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static ClawGraph()
Returns a claw graph.

A claw graph is named for its shape. It is actually a complete bipartite graph with (n1, n2) = (1, 3).

PLOTTING: See CompleteBipartiteGraph.

EXAMPLES: Show a Claw graph

```
sage: (graphs.ClawGraph()).show() # long time
```

Inspect a Claw graph

```
sage: G = graphs.ClawGraph()
sage: G
Claw graph: Graph on 4 vertices
```

static ClebschGraph()
Return the Clebsch graph.

See the Wikipedia article Clebsch_graph for more information.

EXAMPLES:

```
sage: g = graphs.ClebschGraph()
sage: g.automorphism_group().cardinality()
1920
sage: g.girth()
4
sage: g.chromatic_number()
4
sage: g.diameter()
2
sage: g.show(figsize=[10, 10]) # long time
```

static CompleteBipartiteGraph(*n1*, *n2*, *set_position=True*)
Return a Complete Bipartite Graph on $n1 + n2$ vertices.

A Complete Bipartite Graph is a graph with its vertices partitioned into two groups, $V_1 = \{0, ..., n1 - 1\}$ and $V_2 = \{n1, ..., n1 + n2 - 1\}$. Each $u \in V_1$ is connected to every $v \in V_2$.

INPUT:

- `n1, n2` – number of vertices in each side
- `set_position` – boolean (default `True`); if set to `True`, we assign positions to the vertices so that the set of cardinality $n1$ is on the line $y = 1$ and the set of cardinality $n2$ is on the line $y = 0$.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each complete bipartite graph will be displayed with the first $n1$ nodes on the top row (at $y = 1$) from left to right. The remaining $n2$ nodes appear at $y = 0$, also from left to right. The shorter row (partition with fewer nodes) is stretched to the same length as the longer row, unless the shorter row has 1 node; in which case it is centered. The x values in the plot are in domain $[0, \max(n1, n2)]$.

In the Complete Bipartite graph, there is a visual difference in using the spring-layout algorithm vs. the position dictionary used in this constructor. The position dictionary flattens the graph and separates the partitioned nodes, making it clear which nodes an edge is connected to. The Complete Bipartite graph plotted with the spring-layout algorithm tends to center the nodes in n1 (see spring_med in examples below), thus overlapping its nodes and edges, making it typically hard to decipher.

Filling the position dictionary in advance adds $O(n)$ to the constructor. Feel free to race the constructors below in the examples section. The much larger difference is the time added by the spring-layout algorithm

when plotting. (Also shown in the example below). The spring model is typically described as $O(n^3)$, as appears to be the case in the NetworkX source code.

EXAMPLES:

Two ways of constructing the complete bipartite graph, using different layout algorithms:

```
sage: import networkx
sage: n = networkx.complete_bipartite_graph(389, 157); spring_big = Graph(n)
↪ # long time
sage: posdict_big = graphs.CompleteBipartiteGraph(389, 157)
↪ # long time
```

Compare the plotting:

```
sage: n = networkx.complete_bipartite_graph(11, 17)
sage: spring_med = Graph(n)
sage: posdict_med = graphs.CompleteBipartiteGraph(11, 17)
```

Notice here how the spring-layout tends to center the nodes of $n1$:

```
sage: spring_med.show() # long time
sage: posdict_med.show() # long time
```

View many complete bipartite graphs with a Sage Graphics Array, with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CompleteBipartiteGraph(i+1,4)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

We compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.complete_bipartite_graph(i+1,4)
....:     k = Graph(spr)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

trac ticket #12155:

```
sage: graphs.CompleteBipartiteGraph(5,6).complement()
complement(Complete bipartite graph of order 5+6): Graph on 11 vertices
```

static CompleteGraph(*n*)

Return a complete graph on n nodes.

A Complete Graph is a graph in which all nodes are connected to all other nodes.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each complete graph will be displayed with the first (0) node at the top, with the rest following in a counterclockwise manner.

In the complete graph, there is a big difference visually in using the spring-layout algorithm vs. the position dictionary used in this constructor. The position dictionary flattens the graph, making it clear which nodes an edge is connected to. But the complete graph offers a good example of how the spring-layout works. The edges push outward (everything is connected), causing the graph to appear as a 3-dimensional pointy ball. (See examples below).

EXAMPLES: We view many Complete graphs with a Sage Graphics Array, first with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CompleteGraph(i+3)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

We compare to plotting with the spring-layout algorithm:

```
sage: import networkx
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.complete_graph(i+3)
....:     k = Graph(spr)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

Compare the constructors (results will vary)

```
sage: import networkx
sage: t = cputime()
sage: n = networkx.complete_graph(389); spring389 = Graph(n)
sage: cputime(t)            # random
0.59203700000000126
```

(continues on next page)

(continued from previous page)

```
sage: t = cputime()
sage: posdict389 = graphs.CompleteGraph(389)
sage: cputime(t)            # random
0.6680419999999998
```

We compare plotting:

```
sage: import networkx
sage: n = networkx.complete_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.CompleteGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

static CompleteMultipartiteGraph(l)

Returns a complete multipartite graph.

INPUT:

- `l` – a list of integers : the respective sizes of the components.

EXAMPLES:

A complete tripartite graph with sets of sizes $5, 6, 8$:

```
sage: g = graphs.CompleteMultipartiteGraph([5, 6, 8]); g
Multipartite Graph with set sizes [5, 6, 8]: Graph on 19 vertices
```

It clearly has a chromatic number of 3:

```
sage: g.chromatic_number()
3
```

static CossidentePenttilaGraph(q)

Cossidente-Penttila $((q^3+1)(q+1)/2, (q^2+1)(q-1)/2, (q-3)/2, (q-1)^2/2)$-strongly regular graph

For each odd prime power q, one can partition the points of the $O_6^-(q)$-generalized quadrangle $GQ(q, q^2)$ into two parts, so that on any of them the induced subgraph of the point graph of the GQ has parameters as above [CP2005].

Directly following the construction in [CP2005] is not efficient, as one then needs to construct the dual $GQ(q^2, q)$. Thus we describe here a more efficient approach that we came up with, following a suggestion by T.Penttila. Namely, this partition is invariant under the subgroup $H = \Omega_3(q^2) < O_6^-(q)$. We build the appropriate H, which leaves the form $B(X, Y, Z) = XY + Z^2$ invariant, and pick up two orbits of H on the F_q-points. One them is B-isotropic, and we take the representative $(1 : 0 : 0)$. The other one corresponds to the points of $PG(2, q^2)$ that have all the lines on them either missing the conic specified by B, or intersecting the conic in two points. We take $(1 : 1 : e)$ as the representative. It suffices to pick e so that $e^2 + 1$ is not a square in F_{q^2}. Indeed, The conic can be viewed as the union of $\{(0 : 1 : 0)\}$ and $\{(1 : -t^2 : t) | t \in F_{q^2}\}$. The coefficients of a generic line on $(1 : 1 : e)$ are $[1 : -1 - eb : b]$, for $-1 \neq eb$. Thus, to make sure the intersection with the conic is always even, we need that the discriminant of $1 + (1 + eb)t^2 + tb = 0$ never vanishes, and this is if and only if $e^2 + 1$ is not a square. Further, we need to adjust B, by multiplying it by appropriately chosen ν, so that $(1 : 1 : e)$ becomes isotropic under the relative trace norm $\nu B(X, Y, Z) + (\nu B(X, Y, Z))^q$. The latter is used then to define the graph.

INPUT:

- `q` – an odd prime power.

EXAMPLES:

For $q = 3$ one gets Sims-Gewirtz graph.

```
sage: G=graphs.CossidentePenttilaGraph(3)     # optional - gap_packages (grape)
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages (grape)
(56, 10, 0, 2)
```

For $q > 3$ one gets new graphs.

```
sage: G=graphs.CossidentePenttilaGraph(5)     # optional - gap_packages (grape)
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages (grape)
(378, 52, 1, 8)
```

static CoxeterGraph()

Return the Coxeter graph.

See the Wikipedia article Coxeter_graph.

EXAMPLES:

```
sage: g = graphs.CoxeterGraph()
sage: g.automorphism_group().cardinality()
336
sage: g.girth()
7
sage: g.chromatic_number()
3
sage: g.diameter()
4
sage: g.show(figsize=[10, 10]) # long time
```

static CubeConnectedCycle(d)

Return the cube-connected cycle of dimension d.

The cube-connected cycle of order d is the d-dimensional hypercube with each of its vertices replaced by a cycle of length d. This graph has order $d \times 2^d$. The construction is as follows: Construct vertex (x, y) for $0 \leq x < 2^d$, $0 \leq y < d$. For each vertex, (x, y), add an edge between it and $(x, (y - 1) \mod d)$, $(x, (y + 1) \mod d)$, and $(x \oplus 2^y, y)$, where $\oplus$ is the bitwise xor operator.

For $d = 1$ and 2, the cube-connected cycle graph contains self-loops or multiple edges between a pair of vertices, but for all other d, it is simple.

INPUT:

- `d` – The dimension of the desired hypercube as well as the length of the cycle to be placed at each vertex of the d-dimensional hypercube. d must be a positive integer.

EXAMPLES:

The order of the graph is $d \times 2^d$

```
sage: d = 3
sage: g = graphs.CubeConnectedCycle(d)
sage: len(g) == d*2**d
True
```

The diameter of cube-connected cycles for $d > 3$ is $2d + \lfloor \frac{d}{2} \rfloor - 2$

```
sage: d = 4
sage: g = graphs.CubeConnectedCycle(d)
sage: g.diameter() == 2*d+d//2-2
True
```

All vertices have degree 3 when $d > 1$

```
sage: g = graphs.CubeConnectedCycle(5)
sage: all(g.degree(v) == 3 for v in g)
True
```

static CubeGraph(n)

Returns the hypercube in n dimensions.

The hypercube in n dimension is build upon the binary strings on n bits, two of them being adjacent if they differ in exactly one bit. Hence, the distance between two vertices in the hypercube is the Hamming distance.

EXAMPLES:

The distance between 0100110 and 1011010 is 5, as expected

```
sage: g = graphs.CubeGraph(7)
sage: g.distance('0100110','1011010')
5
```

Plot several n-cubes in a Sage Graphics Array

```
sage: g = []
sage: j = []
sage: for i in range(6):
....:  k = graphs.CubeGraph(i+1)
....:  g.append(k)
...
sage: for i in range(2):
....:  n = []
....:  for m in range(3):
....:      n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:  j.append(n)
...
sage: G = graphics_array(j)
sage: G.show(figsize=[6,4]) # long time
```

Use the plot options to display larger n-cubes

```
sage: g = graphs.CubeGraph(9)
sage: g.show(figsize=[12,12],vertex_labels=False, vertex_size=20) # long time
```

AUTHORS:

- Robert Miller

static CycleGraph(n)

Return a cycle graph with n nodes.

A cycle graph is a basic structure which is also typically called an n-gon.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each cycle graph will be displayed with the first (0) node at the top, with the rest following in a counterclockwise manner.

The cycle graph is a good opportunity to compare efficiency of filling a position dictionary vs. using the spring-layout algorithm for plotting. Because the cycle graph is very symmetric, the resulting plots should be similar (in cases of small n).

Filling the position dictionary in advance adds $O(n)$ to the constructor.

EXAMPLES: Compare plotting using the predefined layout and networkx:

```
sage: import networkx
sage: n = networkx.cycle_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.CycleGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

We next view many cycle graphs as a Sage graphics array. First we use the `CycleGraph` constructor, which fills in the position dictionary:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CycleGraph(i+3)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

Compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.cycle_graph(i+3)
....:     k = Graph(spr)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static DegreeSequence(*deg_sequence*)

Returns a graph with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph.

Graph returned is the one returned by the Havel-Hakimi algorithm, which constructs a simple graph by connecting vertices of highest degree to other vertices of highest degree, resorting the remaining vertices by degree and repeating the process. See Theorem 1.4 in [CL1996].

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the degree of a different vertex.

EXAMPLES:

```
sage: G = graphs.DegreeSequence([3,3,3,3])
sage: G.edges(labels=False)
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G.show()  # long time
```

```
sage: G = graphs.DegreeSequence([3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3])
sage: G.show()  # long time
```

```
sage: G = graphs.DegreeSequence([4,4,4,4,4,4,4,4])
sage: G.show()  # long time
```

```
sage: G = graphs.DegreeSequence([1,2,3,4,3,4,3,2,3,2,1])
sage: G.show()  # long time
```

static DegreeSequenceBipartite(*s1*, *s2*)

Returns a bipartite graph whose two sets have the given degree sequences.

Given two different sequences of degrees s_1 and s_2, this functions returns (if possible) a bipartite graph on sets A and B such that the vertices in A have s_1 as their degree sequence, while s_2 is the degree sequence of the vertices in B.

INPUT:

- `s_1` – list of integers corresponding to the degree sequence of the first set.
- `s_2` – list of integers corresponding to the degree sequence of the second set.

ALGORITHM:

This function works through the computation of the matrix given by the Gale-Ryser theorem, which is in this case the adjacency matrix of the bipartite graph.

EXAMPLES:

If we are given as sequences `[2,2,2,2,2]` and `[5,5]` we are given as expected the complete bipartite graph $K_{2,5}$

```
sage: g = graphs.DegreeSequenceBipartite([2,2,2,2,2],[5,5])
sage: g.is_isomorphic(graphs.CompleteBipartiteGraph(5,2))
True
```

Some sequences being incompatible if, for example, their sums are different, the functions raises a `ValueError` when no graph corresponding to the degree sequences exists.

```
sage: g = graphs.DegreeSequenceBipartite([2,2,2,2,1],[5,5])
Traceback (most recent call last):
...
ValueError: There exists no bipartite graph corresponding to the given degree␣
↪sequences
```

static DegreeSequenceConfigurationModel(*deg_sequence*, *seed=None*)

Returns a random pseudograph with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph with multiple edges and loops.

One requirement is that the sum of the degrees must be even, since every edge must be incident with two vertices.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.
- `seed` - a `random.Random` seed or a Python `int` for the random number generator (default: `None`).

EXAMPLES:

```
sage: G = graphs.DegreeSequenceConfigurationModel([1,1])
sage: G.adjacency_matrix()
[0 1]
[1 0]
```

Note: as of this writing, plotting of loops and multiple edges is not supported, and the output is allowed to contain both types of edges.

```
sage: G = graphs.DegreeSequenceConfigurationModel([3,3,3,3,3,3,3,3,3,3,3,3,3,
↪3,3,3,3,3,3,3])
sage: len(G.edges())
30
sage: G.show()  # long time
```

REFERENCE:

[New2003]

static DegreeSequenceExpected(*deg_sequence*, *seed=None*)

Returns a random graph with expected given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph.

One requirement is that the sum of the degrees must be even, since every edge must be incident with two vertices.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.
- `seed` - a `random.Random` seed or a Python `int` for the random number generator (default: `None`).

EXAMPLES:

```
sage: G = graphs.DegreeSequenceExpected([1,2,3,2,3])
sage: G.edges(labels=False)
[(0, 3), (1, 3), (1, 4), (4, 4)]                    # 32-bit
[(0, 3), (1, 4), (2, 2), (2, 3), (2, 4), (4, 4)]    # 64-bit
sage: G.show()  # long time
```

REFERENCE:

[CL2002]

static DegreeSequenceTree(*deg_sequence*)

Returns a tree with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a tree.

Since every tree has one more vertex than edge, the degree sequence must satisfy len(deg_sequence) - sum(deg_sequence)/2 == 1.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.

EXAMPLES:

```
sage: G = graphs.DegreeSequenceTree([3,1,3,3,1,1,1,2,1])
sage: G.show()  # long time
```

static DejterGraph()
Return the Dejter graph.

The Dejter graph is obtained from the binary 7-cube by deleting a copy of the Hamming code of length 7. It is 6-regular, with 112 vertices and 336 edges. For more information, see the Wikipedia article Dejter_graph.

EXAMPLES:

```
sage: g = graphs.DejterGraph(); g
Dejter Graph: Graph on 112 vertices
sage: g.is_regular(k=6)
True
sage: g.girth()
4
```

static DesarguesGraph()
Return the Desargues graph.

PLOTTING: The layout chosen is the same as on the cover of [Har1994].

EXAMPLES:

```
sage: D = graphs.DesarguesGraph()
sage: L = graphs.LCFGraph(20,[5,-5,9,-9],5)
sage: D.is_isomorphic(L)
True
sage: D.show()  # long time
```

static DiamondGraph()
Returns a diamond graph with 4 nodes.

A diamond graph is a square with one pair of diagonal nodes connected.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the diamond graph is drawn as a diamond, with the first node on top, second on the left, third on the right, and fourth on the bottom; with the second and third node connected.

EXAMPLES: Construct and show a diamond graph

```
sage: g = graphs.DiamondGraph()
sage: g.show() # long time
```

static DipoleGraph(*n*)
Returns a dipole graph with n edges.

A dipole graph is a multigraph consisting of 2 vertices connected with n parallel edges.

EXAMPLES:

Construct and show a dipole graph with 13 edges:

```
sage: g = graphs.DipoleGraph(13); g
Dipole graph: Multi-graph on 2 vertices
sage: g.show() # long time
```

static DodecahedralGraph()
Returns a Dodecahedral graph (with 20 nodes)

The dodecahedral graph is cubic symmetric, so the spring-layout algorithm will be very effective for display. It is dual to the icosahedral graph.

PLOTTING: The Dodecahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Dodecahedral graph

```
sage: g = graphs.DodecahedralGraph()
sage: g.show() # long time
```

Create several dodecahedral graphs in a Sage graphics array They will be drawn differently due to the use of the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.DodecahedralGraph()
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static DorogovtsevGoltsevMendesGraph(*n*)

Construct the n-th generation of the Dorogovtsev-Goltsev-Mendes graph.

EXAMPLES:

```
sage: G = graphs.DorogovtsevGoltsevMendesGraph(8)
sage: G.size()
6561
```

REFERENCE:

- [1] Dorogovtsev, S. N., Goltsev, A. V., and Mendes, J. F. F., Pseudofractal scale-free web, Phys. Rev. E 066122 (2002).

static DoubleStarSnark()

Return the double star snark.

The double star snark is a 3-regular graph on 30 vertices. See the Wikipedia article Double-star_snark.

EXAMPLES:

```
sage: g = graphs.DoubleStarSnark()
sage: g.order()
30
sage: g.size()
45
sage: g.chromatic_number()
3
sage: g.is_hamiltonian()
False
sage: g.automorphism_group().cardinality()
80
sage: g.show()
```

static DurerGraph()
Return the Dürer graph.

For more information, see the Wikipedia article D%C3%BCrer_graph.

EXAMPLES:

The Dürer graph is named after Albrecht Dürer. It is a planar graph with 12 vertices and 18 edges.

```
sage: G = graphs.DurerGraph(); G
Durer graph: Graph on 12 vertices
sage: G.is_planar()
True
sage: G.order()
12
sage: G.size()
18
```

The Dürer graph has chromatic number 3, diameter 4, and girth 3.

```
sage: G.chromatic_number()
3
sage: G.diameter()
4
sage: G.girth()
3
```

Its automorphism group is isomorphic to D_6.

```
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(6))
True
```

static DyckGraph()
Return the Dyck graph.

For more information, see the MathWorld article on the Dyck graph or the Wikipedia article Dyck_graph.

EXAMPLES:

The Dyck graph was defined by Walther von Dyck in 1881. It has 32 vertices and 48 edges, and is a cubic graph (regular of degree 3):

```
sage: G = graphs.DyckGraph(); G
Dyck graph: Graph on 32 vertices
sage: G.order()
32
sage: G.size()
48
sage: G.is_regular()
True
sage: G.is_regular(3)
True
```

It is non-planar and Hamiltonian, as well as bipartite (making it a bicubic graph):

```
sage: G.is_planar()
False
sage: G.is_hamiltonian()
True
```

(continues on next page)

(continued from previous page)

```
sage: G.is_bipartite()
True
```

It has radius 5, diameter 5, and girth 6:

```
sage: G.radius()
5
sage: G.diameter()
5
sage: G.girth()
6
```

Its chromatic number is 2 and its automorphism group is of order 192:

```
sage: G.chromatic_number()
2
sage: G.automorphism_group().cardinality()
192
```

It is a non-integral graph as it has irrational eigenvalues:

```
sage: G.characteristic_polynomial().factor()
(x - 3) * (x + 3) * (x - 1)^9 * (x + 1)^9 * (x^2 - 5)^6
```

It is a toroidal graph, and its embedding on a torus is dual to an embedding of the Shrikhande graph (*ShrikhandeGraph*).

static EgawaGraph(*p*, *s*)

Return the Egawa graph with parameters p, s.

Egawa graphs are a peculiar family of graphs devised by Yoshimi Egawa in [Ega1981] . The Shrikhande graph is a special case of this family of graphs, with parameters $(1, 0)$. All the graphs in this family are not recognizable by 1-WL (Weisfeiler Lehamn algorithm of the first order) and 2-WL, that is their orbits are not correctly returned by k-WL for k lower than 3.

Furthermore, all the graphs in this family are distance-regular, but they are not distance-transitive if $p \neq 0$.

The Egawa graph with parameters $(0, s)$ is isomorphic to the Hamming graph with parameters $(s, 4)$, when the underlying set of the Hamming graph is $[0, 1, 2, 3]$

INPUT:

- **p – power to which the graph named Y in the reference** provided above will be raised
- **s – power to which the graph named X in the reference** provided above will be raised

OUTPUT:

- G – The Egawa graph with parameters (p,s)

EXAMPLES:

Every Egawa graph is distance regular.

```
sage: g = graphs.EgawaGraph(1, 2)
sage: g.is_distance_regular()
True
```

An Egawa graph with parameters (0,s) is isomorphic to the Hamming graph with parameters (s, 4).

```
sage: g = graphs.EgawaGraph(0, 4)
sage: g.is_isomorphic(graphs.HammingGraph(4,4))
True
```

static EllinghamHorton54Graph()

Return the Ellingham-Horton 54-graph.

For more information, see the Wikipedia article Ellingham-Horton_graph.

EXAMPLES:

This graph is 3-regular:

```
sage: g = graphs.EllinghamHorton54Graph()
sage: g.is_regular(k=3)
True
```

It is 3-connected and bipartite:

```
sage: g.vertex_connectivity() # not tested - too long
3
sage: g.is_bipartite()
True
```

It is not Hamiltonian:

```
sage: g.is_hamiltonian() # not tested - too long
False
```

... and it has a nice drawing

```
sage: g.show(figsize=[10, 10]) # not tested - too long
```

static EllinghamHorton78Graph()

Return the Ellingham-Horton 78-graph.

For more information, see the Wikipedia article Ellingham%E2%80%93Horton_graph

EXAMPLES:

This graph is 3-regular:

```
sage: g = graphs.EllinghamHorton78Graph()
sage: g.is_regular(k=3)
True
```

It is 3-connected and bipartite:

```
sage: g.vertex_connectivity() # not tested - too long
3
sage: g.is_bipartite()
True
```

It is not Hamiltonian:

```
sage: g.is_hamiltonian() # not tested - too long
False
```

... and it has a nice drawing

```
sage: g.show(figsize=[10,10]) # not tested - too long
```

static EmptyGraph()

Returns an empty graph (0 nodes and 0 edges).

This is useful for constructing graphs by adding edges and vertices individually or in a loop.

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES: Add one vertex to an empty graph and then show:

```
sage: empty1 = graphs.EmptyGraph()
sage: empty1.add_vertex()
0
sage: empty1.show() # long time
```

Use for loops to build a graph from an empty graph:

```
sage: empty2 = graphs.EmptyGraph()
sage: for i in range(5):
....:     empty2.add_vertex() # add 5 nodes, labeled 0-4
0
1
2
3
4
sage: for i in range(3):
....:     empty2.add_edge(i,i+1) # add edges {[0:1],[1:2],[2:3]}
sage: for i in range(1, 4):
....:     empty2.add_edge(4,i) # add edges {[1:4],[2:4],[3:4]}
sage: empty2.show() # long time
```

static ErreraGraph()

Return the Errera graph.

For more information, see the Wikipedia article Errera_graph.

EXAMPLES:

The Errera graph is named after Alfred Errera. It is a planar graph on 17 vertices and having 45 edges.

```
sage: G = graphs.ErreraGraph(); G
Errera graph: Graph on 17 vertices
sage: G.is_planar()
True
sage: G.order()
17
sage: G.size()
45
```

The Errera graph is Hamiltonian with radius 3, diameter 4, girth 3, and chromatic number 4.

```
sage: G.is_hamiltonian()
True
sage: G.radius()
3
sage: G.diameter()
4
```

(continues on next page)

(continued from previous page)

```
sage: G.girth()
3
sage: G.chromatic_number()
4
```

Each vertex degree is either 5 or 6. That is, if f counts the number of vertices of degree 5 and s counts the number of vertices of degree 6, then $f + s$ is equal to the order of the Errera graph.

```
sage: D = G.degree_sequence()
sage: D.count(5) + D.count(6) == G.order()
True
```

The automorphism group of the Errera graph is isomorphic to the dihedral group of order 20.

```
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(10))
True
```

static EuropeMap(*continental=False*, *year=2018*)

Return European states as a graph of common border.

"European state" here is defined as an independent state having the capital city in Europe. The graph has an edge between those countries that have common *land* border.

INPUT:

- `continental`, a Boolean – if set, only return states in the continental Europe
- `year` – reserved for future use

EXAMPLES:

```
sage: Europe = graphs.EuropeMap(); Europe
Europe Map: Graph on 44 vertices
sage: Europe.neighbors('Ireland')
['United Kingdom']

sage: cont_Europe = graphs.EuropeMap(continental=True)
sage: cont_Europe.order()
40
sage: 'Iceland' in cont_Europe
False
```

static F26AGraph()

Return the F26A graph.

The F26A graph is a symmetric bipartite cubic graph with 26 vertices and 39 edges. For more information, see the Wikipedia article F26A_graph.

EXAMPLES:

```
sage: g = graphs.F26AGraph(); g
F26A Graph: Graph on 26 vertices
sage: g.order(),g.size()
(26, 39)
sage: g.automorphism_group().cardinality()
78
sage: g.girth()
```

(continues on next page)

(continued from previous page)

```
6
sage: g.is_bipartite()
True
sage: g.characteristic_polynomial().factor()
(x - 3) * (x + 3) * (x^4 - 5*x^2 + 3)^6
```

static FibonacciTree(*n*)

Return the graph of the Fibonacci Tree F_i of order n.

The Fibonacci tree F_i is recursively defined as the tree with a root vertex and two attached child trees F_{i-1} and F_{i-2}, where F_1 is just one vertex and F_0 is empty.

INPUT:

- `n` - the recursion depth of the Fibonacci Tree

EXAMPLES:

```
sage: g = graphs.FibonacciTree(3)
sage: g.is_tree()
True
```

```
sage: l1 = [ len(graphs.FibonacciTree(_)) + 1 for _ in range(6) ]
sage: l2 = list(fibonacci_sequence(2,8))
sage: l1 == l2
True
```

AUTHORS:

- Harald Schilly and Yann Laigle-Chapuy (2010-03-25)

static FlowerSnark()

Return a Flower Snark.

A flower snark has 20 vertices. It is part of the class of biconnected cubic graphs with edge chromatic number = 4, known as snarks. (i.e.: the Petersen graph). All snarks are not Hamiltonian, non-planar and have Petersen graph graph minors. See the Wikipedia article Flower_snark.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the nodes are drawn 0-14 on the outer circle, and 15-19 in an inner pentagon.

EXAMPLES: Inspect a flower snark:

```
sage: F = graphs.FlowerSnark()
sage: F
Flower Snark: Graph on 20 vertices
sage: F.graph6_string()
'ShCGHC@?GGg@?@?Gp?K??C?CA?G?_G?Cc'
```

Now show it:

```
sage: F.show() # long time
```

static FoldedCubeGraph(*n*)

Returns the folded cube graph of order 2^{n-1}.

The folded cube graph on 2^{n-1} vertices can be obtained from a cube graph on 2^n vertices by merging together opposed vertices. Alternatively, it can be obtained from a cube graph on 2^{n-1} vertices by adding an edge between opposed vertices. This second construction is the one produced by this method.

See the Wikipedia article Folded_cube_graph for more information.

EXAMPLES:

The folded cube graph of order five is the Clebsch graph:

```
sage: fc = graphs.FoldedCubeGraph(5)
sage: clebsch = graphs.ClebschGraph()
sage: fc.is_isomorphic(clebsch)
True
```

static FolkmanGraph()
Return the Folkman graph.

See the Wikipedia article Folkman_graph.

EXAMPLES:

```
sage: g = graphs.FolkmanGraph()
sage: g.order()
20
sage: g.size()
40
sage: g.diameter()
4
sage: g.girth()
4
sage: g.charpoly().factor()
(x - 4) * (x + 4) * x^10 * (x^2 - 6)^4
sage: g.chromatic_number()
2
sage: g.is_eulerian()
True
sage: g.is_hamiltonian()
True
sage: g.is_vertex_transitive()
False
sage: g.is_bipartite()
True
```

static FosterGraph()
Return the Foster graph.

See the Wikipedia article Foster_graph.

EXAMPLES:

```
sage: g = graphs.FosterGraph()
sage: g.order()
90
sage: g.size()
135
sage: g.diameter()
8
sage: g.girth()
10
sage: g.automorphism_group().cardinality()
4320
sage: g.is_hamiltonian()
True
```

static FranklinGraph()
Return the Franklin graph.

For more information, see the Wikipedia article Franklin_graph.

EXAMPLES:

The Franklin graph is named after Philip Franklin. It is a 3-regular graph on 12 vertices and having 18 edges.

```
sage: G = graphs.FranklinGraph(); G
Franklin graph: Graph on 12 vertices
sage: G.is_regular(3)
True
sage: G.order()
12
sage: G.size()
18
```

The Franklin graph is a Hamiltonian, bipartite graph with radius 3, diameter 3, and girth 4.

```
sage: G.is_hamiltonian()
True
sage: G.is_bipartite()
True
sage: G.radius()
3
sage: G.diameter()
3
sage: G.girth()
4
```

It is a perfect, triangle-free graph having chromatic number 2.

```
sage: G.is_perfect()
True
sage: G.is_triangle_free()
True
sage: G.chromatic_number()
2
```

static FriendshipGraph(n)
Return the friendship graph F_n.

The friendship graph is also known as the Dutch windmill graph. Let C_3 be the cycle graph on 3 vertices. Then F_n is constructed by joining $n \geq 1$ copies of C_3 at a common vertex. If $n = 1$, then F_1 is isomorphic to C_3 (the triangle graph). If $n = 2$, then F_2 is the butterfly graph, otherwise known as the bowtie graph. For more information, see the Wikipedia article Friendship_graph.

INPUT:

- n – positive integer; the number of copies of C_3 to use in constructing F_n.

OUTPUT:

- The friendship graph F_n obtained from n copies of the cycle graph C_3.

See also:

- *GraphGenerators.ButterflyGraph()*

EXAMPLES:

The first few friendship graphs.

```
sage: A = []; B = []
sage: for i in range(9):
....:     g = graphs.FriendshipGraph(i + 1)
....:     A.append(g)
sage: for i in range(3):
....:     n = []
....:     for j in range(3):
....:         n.append(A[3*i + j].plot(vertex_size=20, vertex_labels=False))
....:     B.append(n)
sage: G = graphics_array(B)
sage: G.show()  # long time
```

For $n = 1$, the friendship graph F_1 is isomorphic to the cycle graph C_3, whose visual representation is a triangle.

```
sage: G = graphs.FriendshipGraph(1); G
Friendship graph: Graph on 3 vertices
sage: G.show()  # long time
sage: G.is_isomorphic(graphs.CycleGraph(3))
True
```

For $n = 2$, the friendship graph F_2 is isomorphic to the butterfly graph, otherwise known as the bowtie graph.

```
sage: G = graphs.FriendshipGraph(2); G
Friendship graph: Graph on 5 vertices
sage: G.is_isomorphic(graphs.ButterflyGraph())
True
```

If $n \geq 1$, then the friendship graph F_n has $2n + 1$ vertices and $3n$ edges. It has radius 1, diameter 2, girth 3, and chromatic number 3. Furthermore, F_n is planar and Eulerian.

```
sage: n = randint(1, 10^3)
sage: G = graphs.FriendshipGraph(n)
sage: G.order() == 2*n + 1
True
sage: G.size() == 3*n
True
sage: G.radius()
1
sage: G.diameter()
2
sage: G.girth()
3
sage: G.chromatic_number()
3
sage: G.is_planar()
True
sage: G.is_eulerian()
True
```

static FruchtGraph()

Return a Frucht Graph.

A Frucht graph has 12 nodes and 18 edges. It is the smallest cubic identity graph. It is planar and it is

Hamiltonian. See the Wikipedia article Frucht_graph.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the first seven nodes are on the outer circle, with the next four on an inner circle and the last in the center.

EXAMPLES:

```
sage: FRUCHT = graphs.FruchtGraph()
sage: FRUCHT
Frucht graph: Graph on 12 vertices
sage: FRUCHT.graph6_string()
'KhCKM?_EGK?L'
sage: (graphs.FruchtGraph()).show() # long time
```

static FurerGadget(*k*, *prefix=None*)

Return a Furer gadget of order `k` and their coloring.

Construct the Furer gadget described in [CFI1992], a graph composed by a middle layer of $2^(k-1)$ nodes and two sets of nodes $(a_0, ..., a_{k-1})$ and $(b_0, ..., b_{k-1})$. Each node in the middle is connected to either a_i or b_i, for each i in [0,k[. To read about the complete construction, see [CFI1992]. The returned coloring colors the middle section with one color, and then each pair (a_i, b_i) with another color. Since this method is mainly used to create Furer gadgets for the Cai-Furer-Immerman construction, returning gadgets that don't always have the same vertex labels is important, that's why there is a parameter to manually set a prefix to be appended to each vertex label.

INPUT:

- `k` – The order of the returned Furer gadget, greater than 0.
- **`prefix` – Prefix of to be appended to each vertex label,** so as to individualise the returned Furer gadget. Must be comparable for equality and hashable.

OUTPUT:

- `G` – The Furer gadget of order `k`
- **`coloring` – A list of list of vertices, representing the** partition induced by the coloring of `G`'s vertices

EXAMPLES:

Furer gadget of order 3, without any prefix.

```
sage: G, p = graphs.FurerGadget(3)
sage: sorted(G, key=str)
[(), (0, 'a'), (0, 'b'), (0, 1), (0, 2),
 (1, 'a'), (1, 'b'), (1, 2), (2, 'a'), (2, 'b')]
sage: sorted(G.edge_iterator(), key=str)
[((), (0, 'b'), None), ((), (1, 'b'), None),
 ((), (2, 'b'), None), ((0, 'b'), (1, 2), None),
 ((0, 1), (0, 'a'), None), ((0, 1), (1, 'a'), None),
 ((0, 1), (2, 'b'), None), ((0, 2), (0, 'a'), None),
 ((0, 2), (1, 'b'), None), ((0, 2), (2, 'a'), None),
 ((1, 2), (1, 'a'), None), ((1, 2), (2, 'a'), None)]
```

Furer gadget of order 3, with a prefix.

```
sage: G, p = graphs.FurerGadget(3, 'Prefix')
sage: sorted(G, key=str)
[('Prefix', ()), ('Prefix', (0, 'a')), ('Prefix', (0, 'b')),
```

(continues on next page)

(continued from previous page)

```
 ('Prefix', (0, 1)), ('Prefix', (0, 2)), ('Prefix', (1, 'a')),
 ('Prefix', (1, 'b')), ('Prefix', (1, 2)), ('Prefix', (2, 'a')),
 ('Prefix', (2, 'b'))]
sage: sorted(G.edge_iterator(), key=str)
[(('Prefix', ()), ('Prefix', (0, 'b')), None),
 (('Prefix', ()), ('Prefix', (1, 'b')), None),
 (('Prefix', ()), ('Prefix', (2, 'b')), None),
 (('Prefix', (0, 'b')), ('Prefix', (1, 2)), None),
 (('Prefix', (0, 1)), ('Prefix', (0, 'a')), None),
 (('Prefix', (0, 1)), ('Prefix', (1, 'a')), None),
 (('Prefix', (0, 1)), ('Prefix', (2, 'b')), None),
 (('Prefix', (0, 2)), ('Prefix', (0, 'a')), None),
 (('Prefix', (0, 2)), ('Prefix', (1, 'b')), None),
 (('Prefix', (0, 2)), ('Prefix', (2, 'a')), None),
 (('Prefix', (1, 2)), ('Prefix', (1, 'a')), None),
 (('Prefix', (1, 2)), ('Prefix', (2, 'a')), None)]
```

static FuzzyBallGraph(*partition*, *q*)

Construct a Fuzzy Ball graph with the integer partition `partition` and `q` extra vertices.

Let q be an integer and let $m_1, m_2, ..., m_k$ be a set of positive integers. Let $n = q + m_1 + ... + m_k$. The Fuzzy Ball graph with partition $m_1, m_2, ..., m_k$ and q extra vertices is the graph constructed from the graph $G = K_n$ by attaching, for each $i = 1, 2, ..., k$, a new vertex a_i to m_i distinct vertices of G.

For given positive integers k and m and nonnegative integer q, the set of graphs `FuzzyBallGraph(p, q)` for all partitions p of m with k parts are cospectral with respect to the normalized Laplacian.

EXAMPLES:

```
sage: F = graphs.FuzzyBallGraph([3,1],2)
sage: F.adjacency_matrix(vertices=list(F))
[0 0 1 1 1 0 0 0]
[0 0 0 0 0 1 0 0]
[1 0 0 1 1 1 1 1]
[1 0 1 0 1 1 1 1]
[1 0 1 1 0 1 1 1]
[0 1 1 1 1 0 1 1]
[0 0 1 1 1 1 0 1]
[0 0 1 1 1 1 1 0]
```

Pick positive integers m and k and a nonnegative integer q. All the FuzzyBallGraphs constructed from partitions of m with k parts should be cospectral with respect to the normalized Laplacian:

```
sage: m=4; q=2; k=2
sage: g_list=[graphs.FuzzyBallGraph(p,q) for p in Partitions(m, length=k)]
sage: set([g.laplacian_matrix(normalized=True, vertices=list(g)).charpoly()␣
↪for g in g_list])  # long time (7s on sage.math, 2011)
{x^8 - 8*x^7 + 4079/150*x^6 - 68689/1350*x^5 + 610783/10800*x^4 - 120877/
↪3240*x^3 + 1351/100*x^2 - 931/450*x}
```

static GeneralizedPetersenGraph(*n*, *k*)

Returns a generalized Petersen graph with $2n$ nodes. The variables n, k are integers such that $n > 2$ and $0 < k \leq \lfloor (n-1)/2 \rfloor$

For $k = 1$ the result is a graph isomorphic to the circular ladder graph with the same n. The regular Petersen Graph has $n = 5$ and $k = 2$. Other named graphs that can be described using this notation include the Desargues graph and the Möbius-Kantor graph.

INPUT:

- n - the number of nodes is $2 * n$.
- k - integer $0 < k \leq \lfloor (n-1)/2 \rfloor$. Decides how inner vertices are connected.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the generalized Petersen graphs are displayed as an inner and outer cycle pair, with the first n nodes drawn on the outer circle. The first (0) node is drawn at the top of the outer-circle, moving counterclockwise after that. The inner circle is drawn with the (n)th node at the top, then counterclockwise as well.

EXAMPLES: For $k = 1$ the resulting graph will be isomorphic to a circular ladder graph.

```
sage: g = graphs.GeneralizedPetersenGraph(13,1)
sage: g2 = graphs.CircularLadderGraph(13)
sage: g.is_isomorphic(g2)
True
```

The Desargues graph:

```
sage: g = graphs.GeneralizedPetersenGraph(10,3)
sage: g.girth()
6
sage: g.is_bipartite()
True
```

AUTHORS:

- Anders Jonsson (2009-10-15)

static GoethalsSeidelGraph(k, r)

Returns the graph Goethals-Seidel(k, r).

The graph Goethals-Seidel(k, r) comes from a construction presented in Theorem 2.4 of [GS1970]. It relies on a `(v,k)-BIBD` with r blocks and a `hadamard_matrix()` of order $r + 1$. The result is a *sage.graphs.strongly_regular_db.strongly_regular_graph()* on $v(r + 1)$ vertices with degree $k = (n + r - 1)/2$.

It appears under this name in Andries Brouwer's database of strongly regular graphs.

INPUT:

- k, r – integers

See also:

- *is_goethals_seidel()*

EXAMPLES:

```
sage: graphs.GoethalsSeidelGraph(3,3)
Graph on 28 vertices
sage: graphs.GoethalsSeidelGraph(3,3).is_strongly_regular(parameters=True)
(28, 15, 6, 10)
```

static GoldnerHararyGraph()

Return the Goldner-Harary graph.

For more information, see the Wikipedia article Goldner%E2%80%93Harary_graph.

EXAMPLES:

The Goldner-Harary graph is named after A. Goldner and Frank Harary. It is a planar graph having 11 vertices and 27 edges.

```
sage: G = graphs.GoldnerHararyGraph(); G
Goldner-Harary graph: Graph on 11 vertices
sage: G.is_planar()
True
sage: G.order()
11
sage: G.size()
27
```

The Goldner-Harary graph is chordal with radius 2, diameter 2, and girth 3.

```
sage: G.is_chordal()
True
sage: G.radius()
2
sage: G.diameter()
2
sage: G.girth()
3
```

Its chromatic number is 4 and its automorphism group is isomorphic to the dihedral group D_6.

```
sage: G.chromatic_number()
4
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(6))
True
```

static GolombGraph()

Return the Golomb graph.

See the Wikipedia article Golomb_graph for more information.

EXAMPLES:

The Golomb graph is a planar and Hamiltonian graph with 10 vertices and 18 edges. It has chromatic number 4, diameter 3, radius 2 and girth 3. It can be drawn in the plane as a unit distance graph:

```
sage: G = graphs.GolombGraph(); G
Golomb graph: Graph on 10 vertices
sage: pos = G.get_pos()
sage: dist2 = lambda u,v:(u[0]-v[0])**2 + (u[1]-v[1])**2
sage: all(dist2(pos[u], pos[v]) == 1 for u, v in G.edge_iterator(labels=None))
True
```

static GossetGraph()

Return the Gosset graph.

The Gosset graph is the skeleton of the `Gosset_3_21()` polytope. It has with 56 vertices and degree 27. For more information, see the Wikipedia article Gosset_graph.

EXAMPLES:

```
sage: g = graphs.GossetGraph(); g
Gosset Graph: Graph on 56 vertices
```

(continues on next page)

(continued from previous page)

```
sage: g.order(), g.size()
(56, 756)
```

static GrayGraph(*embedding=1*)

Return the Gray graph.

See the Wikipedia article Gray_graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.GrayGraph()
sage: g.order()
54
sage: g.size()
81
sage: g.girth()
8
sage: g.diameter()
6
sage: g.show(figsize=[10, 10])    # long time
sage: graphs.GrayGraph(embedding = 2).show(figsize=[10, 10])    # long time
```

static Grid2dGraph(*n1*, *n2*, *set_positions=True*)

Returns a 2-dimensional grid graph with $n_1 n_2$ nodes (n_1 rows and n_2 columns).

A 2d grid graph resembles a 2 dimensional grid. All inner nodes are connected to their 4 neighbors. Outer (non-corner) nodes are connected to their 3 neighbors. Corner nodes are connected to their 2 neighbors.

INPUT:

- `n1` and `n2` – two positive integers
- `set_positions` – (default: `True`) boolean use to prevent setting the position of the nodes.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, nodes are labelled in (row, column) pairs with $(0, 0)$ in the top left corner. Edges will always be horizontal and vertical - another advantage of filling the position dictionary.

EXAMPLES: Construct and show a grid 2d graph Rows = 5, Columns = 7

```
sage: g = graphs.Grid2dGraph(5,7)
sage: g.show() # long time
```

static GridGraph(*dim_list*)

Returns an n-dimensional grid graph.

INPUT:

- `dim_list` - a list of integers representing the number of nodes to extend in each dimension.

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES:

```
sage: G = graphs.GridGraph([2,3,4])
sage: G.show()  # long time
```

```
sage: C = graphs.CubeGraph(4)
sage: G = graphs.GridGraph([2,2,2,2])
sage: C.show()  # long time
sage: G.show()  # long time
```

static GrotzschGraph()

Return the Grötzsch graph.

The Grötzsch graph is an example of a triangle-free graph with chromatic number equal to 4. For more information, see the Wikipedia article Gr%C3%B6tzsch_graph.

EXAMPLES:

The Grötzsch graph is named after Herbert Grötzsch. It is a Hamiltonian graph with 11 vertices and 20 edges.

```
sage: G = graphs.GrotzschGraph(); G
Grotzsch graph: Graph on 11 vertices
sage: G.is_hamiltonian()
True
sage: G.order()
11
sage: G.size()
20
```

The Grötzsch graph is triangle-free and having radius 2, diameter 2, and girth 4.

```
sage: G.is_triangle_free()
True
sage: G.radius()
2
sage: G.diameter()
2
sage: G.girth()
4
```

Its chromatic number is 4 and its automorphism group is isomorphic to the dihedral group D_5.

```
sage: G.chromatic_number()
4
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(5))
True
```

static HaemersGraph(*q*, *hyperoval=None*, *hyperoval_matching=None*, *field=None*, *check_hyperoval=True*)

Return the Haemers graph obtained from $T_2^*(q)^*$

Let q be a power of 2. In Sect. 8.A of [BL1984] one finds a construction of a strongly regular graph with parameters $(q^2(q+2), q^2+q-1, q-2, q)$ from the graph of $T_2^*(q)^*$, constructed by `T2starGeneralizedQuadrangleGraph()`, by redefining adjacencies in the way specified by an arbitrary `hyperoval_matching` of the points (i.e. partitioning into size two parts) of `hyperoval` defining $T_2^*(q)^*$.

While [BL1984] gives the construction in geometric terms, it can be formulated, and is implemented, in graph-theoretic ones, of re-adjusting the edges. Namely, $G = T_2^*(q)^*$ has a partition into $q+2$ independent sets I_k of size q^2 each. Each vertex in I_j is adjacent to q vertices from I_k. Each I_k is paired to some $I_{k'}$, according to `hyperoval_matching`. One adds edges (s,t) for $s,t \in I_k$ whenever s and t are adjacent to some $u \in I_{k'}$, and removes all the edges between I_k and $I_{k'}$.

INPUT:

- q – a power of two
- hyperoval_matching – if None (default), pair each i-th point of hyperoval with $(i+1)$-th. Otherwise, specifies the pairing in the format $((i_1, i'_1), (i_2, i'_2), ...)$.
- hyperoval – a hyperoval defining $T_2^*(q)^*$. If None (default), the classical hyperoval obtained from a conic is used. See the documentation of *T2starGeneralizedQuadrangleGraph()*, for more information.
- field – an instance of a finite field of order q, must be provided if hyperoval is provided.
- check_hyperoval – (default: True) if True, check hyperoval for correctness.

EXAMPLES:

using the built-in constructions:

```
sage: g=graphs.HaemersGraph(4); g
Haemers(4): Graph on 96 vertices
sage: g.is_strongly_regular(parameters=True)
(96, 19, 2, 4)
```

supplying your own hyperoval_matching:

```
sage: g=graphs.HaemersGraph(4,hyperoval_matching=((0,5),(1,4),(2,3))); g
Haemers(4): Graph on 96 vertices
sage: g.is_strongly_regular(parameters=True)
(96, 19, 2, 4)
```

static HallJankoGraph (*from_string=True*)

Return the Hall-Janko graph.

For more information on the Hall-Janko graph, see the Wikipedia article Hall-Janko_graph.

The construction used to generate this graph in Sage is by a 100-point permutation representation of the Janko group J_2, as described in version 3 of the ATLAS of Finite Group representations, in particular on the page ATLAS: J2 – Permutation representation on 100 points.

INPUT:

- from_string (boolean) – whether to build the graph from its sparse6 string or through GAP. The two methods return the same graph though doing it through GAP takes more time. It is set to True by default.

EXAMPLES:

```
sage: g = graphs.HallJankoGraph()
sage: g.is_regular(36)
True
sage: g.is_vertex_transitive()
True
```

Is it really strongly regular with parameters 14, 12?

```
sage: nu = set(g.neighbors(0))
sage: for v in range(1, 100):
....:     if v in nu:
....:         expected = 14
....:     else:
```

(continues on next page)

(continued from previous page)

```
....:         expected = 12
....:     nv = set(g.neighbors(v))
....:     nv.discard(0)
....:     if len(nu & nv) != expected:
....:         print("Something is wrong here!!!")
....:         break
```

Some other properties that we know how to check:

```
sage: g.diameter()
2
sage: g.girth()
3
sage: factor(g.characteristic_polynomial())
(x - 36) * (x - 6)^36 * (x + 4)^63
```

static HammingGraph(*n*, *q*, *X=None*)

Returns the Hamming graph with parameters `n`, `q` over `X`.

Hamming graphs are graphs over the cartesian product of n copies of `X`, where $q = |X|$, where the vertices, labelled with the corresponding tuple in X^n, are connected if the Hamming distance between their labels is 1. All Hamming graphs are regular, vertex-transitive and distance-regular.

Hamming graphs with parameters $(1, q)$ represent the complete graph with q vertices over the set `X`.

INPUT:

- **`n` – power to which `X` will be raised to provide vertices** for the Hamming graph
- `q` – cardinality of `X`
- **`X` – list of labels representing the vertices of the** underlying graph the Hamming graph will be based on; if `None` (or left unused), the list $[0, ..., q-1]$ will be used

OUTPUT:

- `G` – The Hamming graph with parameters (n, q, X)

EXAMPLES:

Every Hamming graph is distance-regular, regular and vertex-transitive.

```
sage: g = graphs.HammingGraph(3, 7)
sage: g.is_distance_regular()
True
sage: g.is_regular()
True
sage: g.is_vertex_transitive()
True
```

A Hamming graph with parameters (1,q) is isomorphic to the Complete graph with parameter q.

```
sage: g = graphs.HammingGraph(1, 23)
sage: g.is_isomorphic(graphs.CompleteGraph(23))
True
```

If a parameter `q` is provided which is not equal to `X`'s cardinality, an exception is raised.

```
sage: X = ['a','b','c','d','e']
sage: g = graphs.HammingGraph(2, 3, X)
Traceback (most recent call last):
...
ValueError: q must be the cardinality of X
```

REFERENCES:

For a more accurate description, see the following wikipedia page: Wikipedia article Hamming_graph

static HanoiTowerGraph(*pegs*, *disks*, *labels=True*, *positions=True*)

Returns the graph whose vertices are the states of the Tower of Hanoi puzzle, with edges representing legal moves between states.

INPUT:

- `pegs` - the number of pegs in the puzzle, 2 or greater
- `disks` - the number of disks in the puzzle, 1 or greater
- `labels` - default: `True`, if `True` the graph contains more meaningful labels, see explanation below. For large instances, turn off labels for much faster creation of the graph.
- `positions` - default: `True`, if `True` the graph contains layout information. This creates a planar layout for the case of three pegs. For large instances, turn off layout information for much faster creation of the graph.

OUTPUT:

The Tower of Hanoi puzzle has a certain number of identical pegs and a certain number of disks, each of a different radius. Initially the disks are all on a single peg, arranged in order of their radii, with the largest on the bottom.

The goal of the puzzle is to move the disks to any other peg, arranged in the same order. The one constraint is that the disks resident on any one peg must always be arranged with larger radii lower down.

The vertices of this graph represent all the possible states of this puzzle. Each state of the puzzle is a tuple with length equal to the number of disks, ordered by largest disk first. The entry of the tuple is the peg where that disk resides. Since disks on a given peg must go down in size as we go up the peg, this totally describes the state of the puzzle.

For example `(2,0,0)` means the large disk is on peg 2, the medium disk is on peg 0, and the small disk is on peg 0 (and we know the small disk must be above the medium disk). We encode these tuples as integers with a base equal to the number of pegs, and low-order digits to the right.

Two vertices are adjacent if we can change the puzzle from one state to the other by moving a single disk. For example, `(2,0,0)` is adjacent to `(2,0,1)` since we can move the small disk off peg 0 and onto (the empty) peg 1. So the solution to a 3-disk puzzle (with at least two pegs) can be expressed by the shortest path between `(0,0,0)` and `(1,1,1)`. For more on this representation of the graph, or its properties, see [AD2010].

For greatest speed we create graphs with integer vertices, where we encode the tuples as integers with a base equal to the number of pegs, and low-order digits to the right. So for example, in a 3-peg puzzle with 5 disks, the state `(1,2,0,1,1)` is encoded as $1 * 3^4 + 2 * 3^3 + 0 * 3^2 + 1 * 3^1 + 1 * 3^0 = 139$.

For smaller graphs, the labels that are the tuples are informative, but slow down creation of the graph. Likewise computing layout information also incurs a significant speed penalty. For maximum speed, turn off labels and layout and decode the vertices explicitly as needed. The `sage.rings.integer.Integer.digits()` with the `padsto` option is a quick way to do this, though you may want to reverse the list that is output.

PLOTTING:

The layout computed when `positions = True` will look especially good for the three-peg case, when the graph is known to be planar. Except for two small cases on 4 pegs, the graph is otherwise not planar, and likely there is a better way to layout the vertices.

EXAMPLES:

A classic puzzle uses 3 pegs. We solve the 5 disk puzzle using integer labels and report the minimum number of moves required. Note that $3^5 - 1$ is the state where all 5 disks are on peg 2.

```
sage: H = graphs.HanoiTowerGraph(3, 5, labels=False, positions=False)
sage: H.distance(0, 3^5-1)
31
```

A slightly larger instance.

```
sage: H = graphs.HanoiTowerGraph(4, 6, labels=False, positions=False)
sage: H.num_verts()
4096
sage: H.distance(0, 4^6-1)
17
```

For a small graph, labels and layout information can be useful. Here we explicitly list a solution as a list of states.

```
sage: H = graphs.HanoiTowerGraph(3, 3, labels=True, positions=True)
sage: H.shortest_path((0,0,0), (1,1,1))
[(0, 0, 0), (0, 0, 1), (0, 2, 1), (0, 2, 2), (1, 2, 2), (1, 2, 0), (1, 1, 0),␣
↪(1, 1, 1)]
```

Some facts about this graph with p pegs and d disks:

- only automorphisms are the "obvious" ones - renumber the pegs.
- chromatic number is less than or equal to p
- independence number is p^{d-1}

```
sage: H = graphs.HanoiTowerGraph(3,4,labels=False,positions=False)
sage: H.automorphism_group().is_isomorphic(SymmetricGroup(3))
True
sage: H.chromatic_number()
3
sage: len(H.independent_set()) == 3^(4-1)
True
```

AUTHOR:

- Rob Beezer, (2009-12-26), with assistance from Su Doree

static HararyGraph(k, n)

Returns the Harary graph on n vertices and connectivity k, where $2 \leq k < n$.

A k-connected graph G on n vertices requires the minimum degree $\delta(G) \geq k$, so the minimum number of edges G should have is $\lceil kn/2 \rceil$. Harary graphs achieve this lower bound, that is, Harary graphs are minimal k-connected graphs on n vertices.

The construction provided uses the method CirculantGraph. For more details, see the book D. B. West, Introduction to Graph Theory, 2nd Edition, Prentice Hall, 2001, p. 150–151; or the MathWorld article on Harary graphs.

EXAMPLES:

Harary graphs $H_{k,n}$:

```
sage: h = graphs.HararyGraph(5,9); h
Harary graph 5, 9: Graph on 9 vertices
sage: h.order()
9
sage: h.size()
23
sage: h.vertex_connectivity()
5
```

static HarborthGraph()

Return the Harborth Graph.

The Harborth graph has 104 edges and 52 vertices, and is the smallest known example of a 4-regular matchstick graph. For more information, see the Wikipedia article Harborth_graph.

EXAMPLES:

```
sage: g = graphs.HarborthGraph(); g
Harborth Graph: Graph on 52 vertices
sage: g.is_regular(4)
True
```

static HarriesGraph(*embedding=1*)

Return the Harries Graph.

The Harries graph is a Hamiltonian 3-regular graph on 70 vertices. See the Wikipedia article Harries_graph.

The default embedding here is to emphasize the graph's 4 orbits. This graph actually has a funny construction. The following procedure gives an idea of it, though not all the adjacencies are being properly defined.

1. Take two disjoint copies of a *Petersen graph*. Their vertices will form an orbit of the final graph.
2. Subdivide all the edges once, to create 15+15=30 new vertices, which together form another orbit.
3. Create 15 vertices, each of them linked to 2 corresponding vertices of the previous orbit, one in each of the two subdivided Petersen graphs. At the end of this step all vertices from the previous orbit have degree 3, and the only vertices of degree 2 in the graph are those that were just created.
4. Create 5 vertices connected only to the ones from the previous orbit so that the graph becomes 3-regular.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.HarriesGraph()
sage: g.order()
70
sage: g.size()
105
sage: g.girth()
10
sage: g.diameter()
6
sage: g.show(figsize=[10, 10])    # long time
sage: graphs.HarriesGraph(embedding=2).show(figsize=[10, 10])    # long time
```

static HarriesWongGraph(*embedding=1*)

Return the Harries-Wong Graph.

See the Wikipedia article Harries-Wong_graph.

About the default embedding:

The default embedding is an attempt to emphasize the graph's 8 (!!!) different orbits. In order to understand this better, one can picture the graph as being built in the following way:

1. One first creates a 3-dimensional cube (8 vertices, 12 edges), whose vertices define the first orbit of the final graph.
2. The edges of this graph are subdivided once, to create 12 new vertices which define a second orbit.
3. The edges of the graph are subdivided once more, to create 24 new vertices giving a third orbit.
4. 4 vertices are created and made adjacent to the vertices of the second orbit so that they have degree 3. These 4 vertices also define a new orbit.
5. In order to make the vertices from the third orbit 3-regular (they all miss one edge), one creates a binary tree on 1 + 3 + 6 + 12 vertices. The leaves of this new tree are made adjacent to the 12 vertices of the third orbit, and the graph is now 3-regular. This binary tree contributes 4 new orbits to the Harries-Wong graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.HarriesWongGraph()
sage: g.order()
70
sage: g.size()
105
sage: g.girth()
10
sage: g.diameter()
6
sage: orbits = g.automorphism_group(orbits=True)[-1] # long time
sage: g.show(figsize=[15, 15], partition=orbits)     # long time
```

Alternative embedding:

```
sage: graphs.HarriesWongGraph(embedding=2).show()
```

static HeawoodGraph()

Return a Heawood graph.

The Heawood graph is a cage graph that has 14 nodes. It is a cubic symmetric graph. (See also the Möbius-Kantor graph). It is nonplanar and Hamiltonian. It has diameter = 3, radius = 3, girth = 6, chromatic number = 2. It is 4-transitive but not 5-transitive. See the Wikipedia article Heawood_graph.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the nodes are positioned in a circular layout with the first node appearing at the top, and then continuing counterclockwise.

EXAMPLES:

```
sage: H = graphs.HeawoodGraph()
sage: H
Heawood graph: Graph on 14 vertices
sage: H.graph6_string()
'MhEGHC@AI?_PC@_G_'
sage: (graphs.HeawoodGraph()).show() # long time
```

static HerschelGraph()

Return the Herschel graph.

For more information, see the Wikipedia article Herschel_graph.

EXAMPLES:

The Herschel graph is named after Alexander Stewart Herschel. It is a planar, bipartite graph with 11 vertices and 18 edges.

```
sage: G = graphs.HerschelGraph(); G
Herschel graph: Graph on 11 vertices
sage: G.is_planar()
True
sage: G.is_bipartite()
True
sage: G.order()
11
sage: G.size()
18
```

The Herschel graph is a perfect graph with radius 3, diameter 4, and girth 4.

```
sage: G.is_perfect()
True
sage: G.radius()
3
sage: G.diameter()
4
sage: G.girth()
4
```

Its chromatic number is 2 and its automorphism group is isomorphic to the dihedral group D_6.

```
sage: G.chromatic_number()
2
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(6))
True
```

static HexahedralGraph()

Returns a hexahedral graph (with 8 nodes).

A regular hexahedron is a 6-sided cube. The hexahedral graph corresponds to the connectivity of the vertices of the hexahedron. This graph is equivalent to a 3-cube.

PLOTTING: The hexahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Hexahedral graph

```
sage: g = graphs.HexahedralGraph()
sage: g.show() # long time
```

Create several hexahedral graphs in a Sage graphics array. They will be drawn differently due to the use of the spring-layout algorithm.

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.HexahedralGraph()
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static HigmanSimsGraph(*relabel=True*)

Return the Higman-Sims graph.

The Higman-Sims graph is a remarkable strongly regular graph of degree 22 on 100 vertices. For example, it can be split into two sets of 50 vertices each, so that each half induces a subgraph isomorphic to the Hoffman-Singleton graph (*HoffmanSingletonGraph()*). This can be done in 352 ways (see Higman-Sims graph by Andries E. Brouwer, accessed 24 October 2009.)

Its most famous property is that the automorphism group has an index 2 subgroup which is one of the 26 sporadic groups. [HS1968]

The construction used here follows [Haf2004].

See also the Wikipedia article Higman–Sims_graph.

INPUT:

- `relabel` - default: `True`. If `True` the vertices will be labeled with consecutive integers. If `False` the labels are strings that are three digits long. "xyz" means the vertex is in group x (zero through three), pentagon or pentagram y (zero through four), and is vertex z (zero through four) of that pentagon or pentagram. See [Haf2004] for more.

OUTPUT:

The Higman-Sims graph.

EXAMPLES:

A split into the first 50 and last 50 vertices will induce two copies of the Hoffman-Singleton graph, and we illustrate another such split, which is obvious based on the construction used.

```
sage: H = graphs.HigmanSimsGraph()
sage: A = H.subgraph(range(0,50))
sage: B = H.subgraph(range(50,100))
sage: K = graphs.HoffmanSingletonGraph()
sage: K.is_isomorphic(A) and K.is_isomorphic(B)
True
sage: C = H.subgraph(range(25,75))
sage: D = H.subgraph(list(range(0,25))+list(range(75,100)))
sage: K.is_isomorphic(C) and K.is_isomorphic(D)
True
```

The automorphism group contains only one nontrivial proper normal subgroup, which is of index 2 and is simple. It is known as the Higman-Sims group.

```
sage: H = graphs.HigmanSimsGraph()
sage: G = H.automorphism_group()
sage: g=G.order(); g
88704000
sage: K = G.normal_subgroups()[1]
sage: K.is_simple()
True
sage: g//K.order()
2
```

AUTHOR:

- Rob Beezer (2009-10-24)

static HoffmanGraph()

Return the Hoffman Graph.

See the Wikipedia article Hoffman_graph.

EXAMPLES:

```
sage: g = graphs.HoffmanGraph()
sage: g.is_bipartite()
True
sage: g.is_hamiltonian() # long time
True
sage: g.radius()
3
sage: g.diameter()
4
sage: g.automorphism_group().cardinality()
48
```

static HoffmanSingletonGraph()

Return the Hoffman-Singleton graph.

The Hoffman-Singleton graph is the Moore graph of degree 7, diameter 2 and girth 5. The Hoffman-Singleton theorem states that any Moore graph with girth 5 must have degree 2, 3, 7 or 57. The first three respectively are the pentagon, the Petersen graph, and the Hoffman-Singleton graph. The existence of a Moore graph with girth 5 and degree 57 is still open.

A Moore graph is a graph with diameter d and girth $2d + 1$. This implies that the graph is regular, and distance regular.

For more details, see [GR2001] and the Wikipedia article Hoffman–Singleton_graph.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. A novel algorithm written by Tom Boothby gives a random layout which is pleasing to the eye.

EXAMPLES:

```
sage: HS = graphs.HoffmanSingletonGraph()
sage: Set(HS.degree())
{7}
sage: HS.girth()
5
sage: HS.diameter()
```

(continues on next page)

(continued from previous page)

```
2
sage: HS.num_verts()
50
```

Note that you get a different layout each time you create the graph.

```
sage: HS.layout()[1]
(-0.844..., 0.535...)
sage: HS = graphs.HoffmanSingletonGraph()
sage: HS.layout()[1]
(-0.904..., 0.425...)
```

static HoltGraph()

Return the Holt graph (also called the Doyle graph).

See the Wikipedia article Holt_graph.

EXAMPLES:

```
sage: g = graphs.HoltGraph();g
Holt graph: Graph on 27 vertices
sage: g.is_regular()
True
sage: g.is_vertex_transitive()
True
sage: g.chromatic_number()
3
sage: g.is_hamiltonian() # long time
True
sage: g.radius()
3
sage: g.diameter()
3
sage: g.girth()
5
sage: g.automorphism_group().cardinality()
54
```

static HortonGraph()

Return the Horton Graph.

The Horton graph is a cubic 3-connected non-hamiltonian graph. For more information, see the Wikipedia article Horton_graph.

EXAMPLES:

```
sage: g = graphs.HortonGraph()
sage: g.order()
96
sage: g.size()
144
sage: g.radius()
10
sage: g.diameter()
10
sage: g.girth()
6
sage: g.automorphism_group().cardinality()
```

(continues on next page)

(continued from previous page)

```
96
sage: g.chromatic_number()
2
sage: g.is_hamiltonian() # not tested -- veeeery long
False
```

static HouseGraph()

Returns a house graph with 5 nodes.

A house graph is named for its shape. It is a triangle (roof) over a square (walls).

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the house graph is drawn with the first node in the lower-left corner of the house, the second in the lower-right corner of the house. The third node is in the upper-left corner connecting the roof to the wall, and the fourth is in the upper-right corner connecting the roof to the wall. The fifth node is the top of the roof, connected only to the third and fourth.

EXAMPLES: Construct and show a house graph

```
sage: g = graphs.HouseGraph()
sage: g.show() # long time
```

static HouseXGraph()

Returns a house X graph with 5 nodes.

A house X graph is a house graph with two additional edges. The upper-right corner is connected to the lower-left. And the upper-left corner is connected to the lower-right.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the house X graph is drawn with the first node in the lower-left corner of the house, the second in the lower-right corner of the house. The third node is in the upper-left corner connecting the roof to the wall, and the fourth is in the upper-right corner connecting the roof to the wall. The fifth node is the top of the roof, connected only to the third and fourth.

EXAMPLES: Construct and show a house X graph

```
sage: g = graphs.HouseXGraph()
sage: g.show() # long time
```

static HyperStarGraph(n, k)

Returns the hyper-star graph HS(n,k).

The vertices of the hyper-star graph are the set of binary strings of length n which contain k 1s. Two vertices, u and v, are adjacent only if u can be obtained from v by swapping the first bit with a different symbol in another position.

INPUT:

- n
- k

EXAMPLES:

```
sage: g = graphs.HyperStarGraph(6,3)
sage: g.plot() # long time
Graphics object consisting of 51 graphics primitives
```

REFERENCES:

- Lee, Hyeong-Ok, Jong-Seok Kim, Eunseuk Oh, and Hyeong-Seok Lim. "Hyper-Star Graph: A New Interconnection Network Improving the Network Cost of the Hypercube." In Proceedings of the First EurAsian Conference on Information and Communication Technology, 858-865. Springer-Verlag, 2002.

AUTHORS:

- Michael Yurko (2009-09-01)

static IcosahedralGraph()

Returns an Icosahedral graph (with 12 nodes).

The regular icosahedron is a 20-sided triangular polyhedron. The icosahedral graph corresponds to the connectivity of the vertices of the icosahedron. It is dual to the dodecahedral graph. The icosahedron is symmetric, so the spring-layout algorithm will be very effective for display.

PLOTTING: The Icosahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show an Octahedral graph

```
sage: g = graphs.IcosahedralGraph()
sage: g.show() # long time
```

Create several icosahedral graphs in a Sage graphics array. They will be drawn differently due to the use of the spring-layout algorithm.

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.IcosahedralGraph()
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static IntersectionGraph(S)

Return the intersection graph of the family S

The intersection graph of a family S is a graph G with $V(G) = S$ such that two elements $s_1, s_2 \in S$ are adjacent in G if and only if $s_1 \cap s_2 \neq \emptyset$.

INPUT:

- S – a list of sets/tuples/iterables

> **Note:** The elements of S must be finite, hashable, and the elements of any $s \in S$ must be hashable too.

EXAMPLES:

```
sage: graphs.IntersectionGraph([(1,2,3),(3,4,5),(5,6,7)])
Intersection Graph: Graph on 3 vertices
```

static IntervalGraph(*intervals*, *points_ordered=False*)

Return the graph corresponding to the given intervals.

An interval graph is built from a list $(a_i, b_i)_{1 \leq i \leq n}$ of intervals : to each interval of the list is associated one vertex, two vertices being adjacent if the two corresponding (closed) intervals intersect.

INPUT:

- `intervals` – the list of pairs (a_i, b_i) defining the graph.
- `points_ordered` – states whether every interval (a_i, b_i) of $intervals$ satisfies $a_i < b_i$. If satisfied then setting `points_ordered` to `True` will speed up the creation of the graph.

Note:

- The vertices are named 0, 1, 2, and so on. The intervals used to create the graph are saved with the graph and can be recovered using `get_vertex()` or `get_vertices()`.

EXAMPLES:

The following line creates the sequence of intervals $(i, i + 2)$ for i in $[0, ..., 8]$:

```
sage: intervals = [(i,i+2) for i in range(9)]
```

In the corresponding graph

```
sage: g = graphs.IntervalGraph(intervals)
sage: g.get_vertex(3)
(3, 5)
sage: neigh = g.neighbors(3)
sage: for v in neigh: print(g.get_vertex(v))
(1, 3)
(2, 4)
(4, 6)
(5, 7)
```

The is_interval() method verifies that this graph is an interval graph.

```
sage: g.is_interval()
True
```

The intervals in the list need not be distinct.

```
sage: intervals = [ (1,2), (1,2), (1,2), (2,3), (3,4) ]
sage: g = graphs.IntervalGraph(intervals,True)
sage: g.clique_maximum()
[0, 1, 2, 3]
sage: g.get_vertices()
{0: (1, 2), 1: (1, 2), 2: (1, 2), 3: (2, 3), 4: (3, 4)}
```

The endpoints of the intervals are not ordered we get the same graph (except for the vertex labels).

```
sage: rev_intervals = [ (2,1), (2,1), (2,1), (3,2), (4,3) ]
sage: h = graphs.IntervalGraph(rev_intervals,False)
sage: h.get_vertices()
{0: (2, 1), 1: (2, 1), 2: (2, 1), 3: (3, 2), 4: (4, 3)}
sage: g.edges() == h.edges()
True
```

static IoninKharaghani765Graph()

Return a $(765, 192, 48, 48)$-strongly regular graph.

Existence of a strongly regular graph with these parameters was claimed in [IK2003]. Implementing the construction in the latter did not work, however. This function implements the following instructions, shared by Yury Ionin and Hadi Kharaghani.

Let A be the affine plane over the field $GF(3) = \{-1, 0, 1\}$. Let

$$\phi_1(x, y) = x$$
$$\phi_2(x, y) = y$$
$$\phi_3(x, y) = x + y$$
$$\phi_4(x, y) = x - y$$

For $i = 1, 2, 3, 4$ and $j \in GF(3)$, let $L_{i,j}$ be the line in A defined by $\phi_i(x, y) = j$. Let $\mathcal{M}$ be the set of all 12 lines $L_{i,j}$, plus the empty set. Let π be the permutation defined on $\mathcal{M}$ by $\pi(L_{i,j}) = L_{i,j+1}$ and $\pi(\emptyset) = \emptyset$, so that π has three orbits of cardinality 3 and one of cardinality 1.

Let $A = (p_1, ..., p_9)$ with $p_1 = (-1, 1)$, $p_2 = (-1, 0)$, $p_3 = (-1, 1)$, $p_4 = (0, -1)$, $p_5 = (0, 0)$, $p_6 = (0, 1)$, $p_7 = (1, -1)$, $p_8 = (1, 0)$, $p_9 = (1, 1)$. Note that $p_i + p_{10-i} = (0, 0)$. For any subset X of A, let $M(X)$ be the $(0, 1)$-matrix of order 9 whose (i, j)-entry equals 1 if and only if $p_{10-i} - p_j \in X$. Note that M is a symmetric matrix.

An MF-tuple is an ordered quintuple $(X_1, X_2, X_3, X_4, X_5)$ of subsets of A, of which one is the empty set and the other four are pairwise non-parallel lines. Such a quintuple generates the following block matrix:

$$N(X_1, X_2, X_3, X_4, X_5) = \begin{pmatrix} M(X_1) & M(X_2) & M(X_3) & M(X_4) & M(X_5) \\ M(X_2) & M(X_3) & M(X_4) & M(X_5) & M(X_1) \\ M(X_3) & M(X_4) & M(X_5) & M(X_1) & M(X_2) \\ M(X_4) & M(X_5) & M(X_1) & M(X_2) & M(X_3) \\ M(X_5) & M(X_1) & M(X_2) & M(X_3) & M(X_4) \end{pmatrix}$$

Observe that if $(X_1, X_2, X_3, X_4, X_5)$ is an MF-tuple, then $N(X_1, X_2, X_3, X_4, X_5)$ is the symmetric incidence matrix of a symmetric $(45, 12, 3)$-design.

Let $\mathcal{F}$ be the set of all MF-tuples and let σ be the following permutation of $\mathcal{F}$:

$$\sigma(X_1, X_2, X_3, X_4, X_5) = (X_2, X_3, X_4, X_5, X_1)$$
$$\pi(X_1, X_2, X_3, X_4, X_5) = (\pi(X_1), \pi(X_2), \pi(X_3), \pi(X_4), \pi(X_5))$$

Observe that σ and π commute, and generate a (cyclic) group G of order 15. We will from now on identify G with the (cyclic) multiplicative group of the field $GF(16)$ equal to $\{\omega^0, ..., \omega^{14}\}$. Let $W = [w_{ij}]$ be the following matrix of order 17 over $GF(16) = \{a_1, ..., a_16\}$:

$$w_{ij} = \begin{cases} a_i + a_j & \text{if } 1 \le i \le 16, 1 \le j \le 16, \\ 1 & \text{if } i = 17, j \ne 17, \\ 1 & \text{if } i \ne 17, j = 17, \\ 0 & \text{if } i = j = 17 \end{cases}$$

The diagonal entries of W are equal to 0, each off-diagonal entry can be represented as ω^k with $0 \le k \le 14$. Matrix W is a symmetric $BGW(17, 16, 15; G)$.

Fix an MF-tuple $(X_1, X_2, X_3, X_4, X_5)$ and let S be the block matrix obtained from W by replacing every diagonal entry of W by the zero matrix of order 45, and every off-diagonal entry ω^k by the matrix $N(\sigma^k(X_1, X_2, X_3, X_4, X_5))$ (through the association of ω^k with an element of G). Then S is a symmetric incidence matrix of a symmetric $(765, 192, 48)$-design with zero diagonal, and therefore S is an adjacency matrix of a strongly regular graph with parameters $(765, 192, 48, 48)$.

EXAMPLES:

```
sage: g = graphs.IoninKharaghani765Graph(); g
Ionin-Kharaghani: Graph on 765 vertices
```

Todo: An update to [IK2003] meant to fix the problem encountered became available 2016/02/24, see http://www.cs.uleth.ca/~hadi/research/IoninKharaghani.pdf

static JankoKharaghaniGraph(*v*)

Return a (936, 375, 150, 150)-srg or a (1800, 1029, 588, 588)-srg.

This functions returns a strongly regular graph for the two sets of parameters shown to be realizable in [JK2002]. The paper also uses a construction from [GM1987].

INPUT:

- v (integer) – one of 936 or 1800.

EXAMPLES:

```
sage: g = graphs.JankoKharaghaniGraph(936)    # long time
sage: g.is_strongly_regular(parameters=True) # long time
(936, 375, 150, 150)

sage: g = graphs.JankoKharaghaniGraph(1800)  # not tested (30s)
sage: g.is_strongly_regular(parameters=True) # not tested (30s)
(1800, 1029, 588, 588)
```

static JankoKharaghaniTonchevGraph()

Return a (324,153,72,72)-strongly regular graph from [JKT2001].

Build the graph using the description given in [JKT2001], taking sets B1 and B163 in the text as adjacencies of vertices 1 and 163, respectively, and taking the edge orbits of the group G provided.

EXAMPLES:

```
sage: Gamma=graphs.JankoKharaghaniTonchevGraph()  # long time
sage: Gamma.is_strongly_regular(parameters=True)  # long time
(324, 153, 72, 72)
```

static JohnsonGraph(n, k)

Returns the Johnson graph with parameters n, k.

Johnson graphs are a special class of undirected graphs defined from systems of sets. The vertices of the Johnson graph $J(n, k)$ are the k-element subsets of an n-element set; two vertices are adjacent when they meet in a $(k - 1)$-element set. See the Wikipedia article Johnson_graph for more information.

EXAMPLES:

The Johnson graph is a Hamiltonian graph.

```
sage: g = graphs.JohnsonGraph(7, 3)
sage: g.is_hamiltonian()
True
```

Every Johnson graph is vertex transitive.

```
sage: g = graphs.JohnsonGraph(6, 4)
sage: g.is_vertex_transitive()
True
```

The complement of the Johnson graph $J(n, 2)$ is isomorphic to the Kneser Graph $K(n, 2)$. In particular the complement of $J(5, 2)$ is isomorphic to the Petersen graph.

```
sage: g = graphs.JohnsonGraph(5,2)
sage: g.complement().is_isomorphic(graphs.PetersenGraph())
True
```

static KingGraph(*dim_list*, *radius=None*, *relabel=False*)

Returns the d-dimensional King Graph with prescribed dimensions.

The 2-dimensional King Graph of parameters n and m is a graph with nm vertices in which each vertex represents a square in an $n \times m$ chessboard, and each edge corresponds to a legal move by a king.

The d-dimensional King Graph with $d >= 2$ has for vertex set the cells of a d-dimensional grid with prescribed dimensions, and each edge corresponds to a legal move by a king in either one or two dimensions.

All 2-dimensional King Graphs are Hamiltonian, biconnected, and have chromatic number 4 as soon as both dimensions are larger or equal to 2.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `radius` – (default: `None`) by setting the radius to a positive integer, one may increase the power of the king to at least `radius` steps. When the radius equals the higher size of the dimensions, the resulting graph is a Queen Graph.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

EXAMPLES:

The $(2, 2)$-King Graph is isomorphic to the complete graph on 4 vertices:

```
sage: G = graphs.QueenGraph( [2, 2] )
sage: G.is_isomorphic( graphs.CompleteGraph(4) )
True
```

The King Graph with large enough radius is isomorphic to a Queen Graph:

```
sage: G = graphs.KingGraph( [5, 4], radius=5 )
sage: H = graphs.QueenGraph( [4, 5] )
sage: G.is_isomorphic( H )
True
```

Also True in higher dimensions:

```
sage: G = graphs.KingGraph( [2, 5, 4], radius=5 )
sage: H = graphs.QueenGraph( [4, 5, 2] )
sage: G.is_isomorphic( H )
True
```

static KittellGraph()

Return the Kittell Graph.

For more information, see the Wolfram page about the Kittel Graph.

EXAMPLES:

```
sage: g = graphs.KittellGraph()
sage: g.order()
23
sage: g.size()
63
sage: g.radius()
3
sage: g.diameter()
4
sage: g.girth()
3
sage: g.chromatic_number()
4
```

static Klein3RegularGraph()

Return the Klein 3-regular graph.

The cubic Klein graph has 56 vertices and can be embedded on a surface of genus 3. It is the dual of *Klein7RegularGraph()*. For more information, see the Wikipedia article Klein_graphs.

EXAMPLES:

```
sage: g = graphs.Klein3RegularGraph(); g
Klein 3-regular Graph: Graph on 56 vertices
sage: g.order(), g.size()
(56, 84)
sage: g.girth()
7
sage: g.automorphism_group().cardinality()
336
sage: g.chromatic_number()
3
```

static Klein7RegularGraph()

Return the Klein 7-regular graph.

The 7-valent Klein graph has 24 vertices and can be embedded on a surface of genus 3. It is the dual of *Klein3RegularGraph()*. For more information, see the Wikipedia article Klein_graphs.

EXAMPLES:

```
sage: g = graphs.Klein7RegularGraph(); g
Klein 7-regular Graph: Graph on 24 vertices
sage: g.order(), g.size()
(24, 84)
sage: g.girth()
3
sage: g.automorphism_group().cardinality()
336
sage: g.chromatic_number()
4
```

static KneserGraph(n, k)

Returns the Kneser Graph with parameters n, k.

The Kneser Graph with parameters n, k is the graph whose vertices are the k-subsets of $[0, 1, \ldots, n-1]$, and such that two vertices are adjacent if their corresponding sets are disjoint.

For example, the Petersen Graph can be defined as the Kneser Graph with parameters $5, 2$.

EXAMPLES:

```
sage: KG = graphs.KneserGraph(5,2)
sage: sorted(KG.vertex_iterator(), key=str)
[{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5},
 {3, 4}, {3, 5}, {4, 5}]
sage: P = graphs.PetersenGraph()
sage: P.is_isomorphic(KG)
True
```

static KnightGraph(*dim_list*, *one=1*, *two=2*, *relabel=False*)

Returns the d-dimensional Knight Graph with prescribed dimensions.

The 2-dimensional Knight Graph of parameters n and m is a graph with nm vertices in which each vertex represents a square in an $n \times m$ chessboard, and each edge corresponds to a legal move by a knight.

The d-dimensional Knight Graph with $d >= 2$ has for vertex set the cells of a d-dimensional grid with prescribed dimensions, and each edge corresponds to a legal move by a knight in any pairs of dimensions.

The (n, n)-Knight Graph is Hamiltonian for even $n > 4$.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `one` – (default: `1`) integer indicating the number on steps in one dimension.
- `two` – (default: `2`) integer indicating the number on steps in the second dimension.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

EXAMPLES:

The $(3, 3)$-Knight Graph has an isolated vertex:

```
sage: G = graphs.KnightGraph( [3, 3] )
sage: G.degree( (1,1) )
0
```

The $(3, 3)$-Knight Graph minus vertex (1,1) is a cycle of order 8:

```
sage: G = graphs.KnightGraph( [3, 3] )
sage: G.delete_vertex( (1,1) )
sage: G.is_isomorphic( graphs.CycleGraph(8) )
True
```

The $(6, 6)$-Knight Graph is Hamiltonian:

```
sage: G = graphs.KnightGraph( [6, 6] )
sage: G.is_hamiltonian()
True
```

static KrackhardtKiteGraph()

Return a Krackhardt kite graph with 10 nodes.

The Krackhardt kite graph was originally developed by David Krackhardt for the purpose of studying social networks (see [Kre2002] and the Wikipedia article Krackhardt_kite_graph). It is used to show the distinction between: degree centrality, betweeness centrality, and closeness centrality. For more information read the plotting section below in conjunction with the example.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the graph is drawn left to right, in top to bottom row sequence of [2, 3, 2, 1, 1, 1] nodes on each row. This places the fourth node (3) in the center of the kite, with the highest degree. But the fourth node only connects nodes that are otherwise connected, or those in its clique (i.e.: Degree Centrality). The eighth (7) node is where the kite meets the tail. It has degree = 3, less than the average, but is the only connection between the kite and tail (i.e.: Betweenness Centrality). The sixth and seventh nodes (5 and 6) are drawn in the third row and have degree = 5. These nodes have the shortest path to all other nodes in the graph (i.e.: Closeness Centrality). Please execute the example for visualization.

EXAMPLES:

Construct and show a Krackhardt kite graph

```
sage: g = graphs.KrackhardtKiteGraph()
sage: g.show() # long time
```

static LCFGraph(*n*, *shift_list*, *repeats*)

Return the cubic graph specified in LCF notation.

LCF (Lederberg-Coxeter-Fruchte) notation is a concise way of describing cubic Hamiltonian graphs. The way a graph is constructed is as follows. Since there is a Hamiltonian cycle, we first create a cycle on n nodes. The variable shift_list = [s_0, s_1, ..., s_k-1] describes edges to be created by the following scheme: for each i, connect vertex i to vertex (i + s_i). Then, repeats specifies the number of times to repeat this process, where on the jth repeat we connect vertex (i + j*len(shift_list)) to vertex (i + j*len(shift_list) + s_i).

INPUT:

- `n` - the number of nodes.
- `shift_list` - a list of integer shifts mod n.
- `repeats` - the number of times to repeat the process.

EXAMPLES:

```
sage: G = graphs.LCFGraph(4, [2,-2], 2)
sage: G.is_isomorphic(graphs.TetrahedralGraph())
True
```

```
sage: G = graphs.LCFGraph(20, [10,7,4,-4,-7,10,-4,7,-7,4], 2)
sage: G.is_isomorphic(graphs.DodecahedralGraph())
True
```

```
sage: G = graphs.LCFGraph(14, [5,-5], 7)
sage: G.is_isomorphic(graphs.HeawoodGraph())
True
```

The largest cubic nonplanar graph of diameter three:

```
sage: G = graphs.LCFGraph(20, [-10,-7,-5,4,7,-10,-7,-4,5,7,-10,-7,6,-5,7,-10,-
↪7,5,-6,7], 1)
sage: G.degree()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
sage: G.diameter()
3
sage: G.show()  # long time
```

PLOTTING: LCF Graphs are plotted as an n-cycle with edges in the middle, as described above.

REFERENCES:

- [1] Frucht, R. "A Canonical Representation of Trivalent Hamiltonian Graphs." J. Graph Th. 1, 45-60, 1976.
- [2] Grunbaum, B. Convex Polytope es. New York: Wiley, pp. 362-364, 1967.
- [3] Lederberg, J. 'DENDRAL-64: A System for Computer Construction, Enumeration and Notation of Organic Molecules as Tree Structures and Cyclic Graphs. Part II. Topology of Cyclic Graphs.' Interim Report to the National Aeronautics and Space Administration. Grant NsG 81-60. December 15, 1965. http://profiles.nlm.nih.gov/BB/A/B/I/U/_/bbabiu.pdf.

static LadderGraph(*n*)

Returns a ladder graph with 2*n nodes.

A ladder graph is a basic structure that is typically displayed as a ladder, i.e.: two parallel path graphs connected at each corresponding node pair.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each ladder graph will be displayed horizontally, with the first n nodes displayed left to right on the top horizontal line.

EXAMPLES: Construct and show a ladder graph with 14 nodes

```
sage: g = graphs.LadderGraph(7)
sage: g.show() # long time
```

Create several ladder graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.LadderGraph(i+2)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static LivingstoneGraph()

Return the Livingstone Graph.

The Livingstone graph is a distance-transitive graph on 266 vertices whose automorphism group is the J1 group. For more information, see the Wikipedia article Livingstone_graph.

EXAMPLES:

```
sage: g = graphs.LivingstoneGraph() # optional - internet
sage: g.order()                     # optional - internet
266
sage: g.size()                      # optional - internet
1463
sage: g.girth()                     # optional - internet
5
sage: g.is_vertex_transitive()      # optional - internet
True
sage: g.is_distance_regular()       # optional - internet
True
```

static LjubljanaGraph(*embedding=1*)
Return the Ljubljana Graph.

The Ljubljana graph is a bipartite 3-regular graph on 112 vertices and 168 edges. It is not vertex-transitive as it has two orbits which are also independent sets of size 56. See the Wikipedia article Ljubljana_graph.

The default embedding is obtained from the Heawood graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.LjubljanaGraph()
sage: g.order()
112
sage: g.size()
168
sage: g.girth()
10
sage: g.diameter()
8
sage: g.show(figsize=[10, 10])    # long time
sage: graphs.LjubljanaGraph(embedding=2).show(figsize=[10, 10])    # long time
```

static LocalMcLaughlinGraph()
Return the local McLaughlin graph.

The local McLaughlin graph is a strongly regular graph with parameters $(162, 56, 10, 24)$. It can be obtained from *McLaughlinGraph()* by considering the stabilizer of a point: one of its orbits has cardinality 162.

EXAMPLES:

```
sage: g = graphs.LocalMcLaughlinGraph(); g    # long time # optional - gap_
↪packages
Local McLaughlin Graph: Graph on 162 vertices
sage: g.is_strongly_regular(parameters=True) # long time # optional - gap_
↪packages
(162, 56, 10, 24)
```

static LollipopGraph(*n1*, *n2*)
Returns a lollipop graph with n1+n2 nodes.

A lollipop graph is a path graph (order n2) connected to a complete graph (order n1). (A barbell graph minus one of the bells).

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the complete graph will be drawn in the lower-left corner with the (n1)th node at a 45 degree angle above the right horizontal center of the complete graph, leading directly into the path graph.

EXAMPLES:

Construct and show a lollipop graph Candy = 13, Stick = 4:

```
sage: g = graphs.LollipopGraph(13,4); g
Lollipop graph: Graph on 17 vertices
sage: g.show() # long time
```

static M22Graph()
Return the M22 graph.

The M_{22} graph is the unique strongly regular graph with parameters $v = 77, k = 16, \lambda = 0, \mu = 4$.

For more information on the M_{22} graph, see https://www.win.tue.nl/~aeb/graphs/M22.html.

EXAMPLES:

```
sage: g = graphs.M22Graph()
sage: g.order()
77
sage: g.size()
616
sage: g.is_strongly_regular(parameters = True)
(77, 16, 0, 4)
```

static MarkstroemGraph()

Return the Markström Graph.

The Markström Graph is a cubic planar graph with no cycles of length 4 nor 8, but containing cycles of length 16. For more information, see the Wolfram page about the Markström Graph.

EXAMPLES:

```
sage: g = graphs.MarkstroemGraph()
sage: g.order()
24
sage: g.size()
36
sage: g.is_planar()
True
sage: g.is_regular(3)
True
sage: g.subgraph_search(graphs.CycleGraph(4)) is None
True
sage: g.subgraph_search(graphs.CycleGraph(8)) is None
True
sage: g.subgraph_search(graphs.CycleGraph(16))
Subgraph of (Markstroem Graph): Graph on 16 vertices
```

static MathonPseudocyclicMergingGraph(*M*, *t*)

Mathon's merging of classes in a pseudo-cyclic 3-class association scheme

Construct strongly regular graphs from p.97 of [BL1984].

INPUT:

- `M` – the list of matrices in a pseudo-cyclic 3-class association scheme. The identity matrix must be the first entry.
- `t` (integer) – the number of the graph, from 0 to 2.

See also:

- *is_muzychuk_S6()*

static MathonPseudocyclicStronglyRegularGraph(*t*, *G=None*, *L=None*)

Return a strongly regular graph on $(4t + 1)(4t - 1)^2$ vertices from [Mat1978].

Let $4t-1$ be a prime power, and $4t+1$ be such that there exists a strongly regular graph G with parameters $(4t+1, 2t, t-1, t)$. In particular, $4t+1$ must be a sum of two squares [Mat1978]. With this input, Mathon [Mat1978] gives a construction of a strongly regular graph with parameters $(4\mu + 1, 2\mu, \mu - 1, \mu)$, where

$\mu = t(4t(4t-1)-1)$. The construction is optionally parametrised by an a skew-symmetric Latin square of order $4t+1$, with entries in $-2t, ..., -1, 0, 1, ..., 2t$.

Our implementation follows a description given in [ST1981].

INPUT:

- `t` – a positive integer
- `G` – if `None` (default), try to construct the necessary graph with parameters $(4t+1, 2t, t-1, t)$, otherwise use the user-supplied one, with vertices labelled from 0 to $4t$.
- `L` – if `None` (default), construct a necessary skew Latin square, otherwise use the user-supplied one. Here non-isomorphic Latin squares – one constructed from $Z/9Z$, and the other from $(Z/3Z)^2$ – lead to non-isomorphic graphs.

See also:

- *is_mathon_PC_srg()*

EXAMPLES:

Using default `G` and `L`.

```
sage: from sage.graphs.generators.families import␣
↪MathonPseudocyclicStronglyRegularGraph
sage: G=MathonPseudocyclicStronglyRegularGraph(1); G
Mathon's PC SRG on 45 vertices: Graph on 45 vertices
sage: G.is_strongly_regular(parameters=True)
(45, 22, 10, 11)
```

Supplying `G` and `L` (constructed from the automorphism group of `G`).

```
sage: G = graphs.PaleyGraph(9)
sage: a = G.automorphism_group(partition=[sorted(G)])
sage: it = (x for x in a.normal_subgroups() if x.order() == 9)
sage: subg = next(iter(it))
sage: r = [matrix(libgap.PermutationMat(libgap(z), 9).sage())
....:       for z in subg]
sage: ff = list(map(lambda y: (y[0]-1,y[1]-1),
....:           Permutation(map(lambda x: 1+r.index(x^-1), r)).cycle_
↪tuples()[1:]))
sage: L = sum(i*(r[a]-r[b]) for i,(a,b) in zip(range(1,len(ff)+1), ff)); L
[ 0  1 -1 -3 -2 -4  3  4  2]
[-1  0  1 -4 -3 -2  2  3  4]
[ 1 -1  0 -2 -4 -3  4  2  3]
[ 3  4  2  0  1 -1 -3 -2 -4]
[ 2  3  4 -1  0  1 -4 -3 -2]
[ 4  2  3  1 -1  0 -2 -4 -3]
[-3 -2 -4  3  4  2  0  1 -1]
[-4 -3 -2  2  3  4 -1  0  1]
[-2 -4 -3  4  2  3  1 -1  0]

sage: G.relabel(range(9))
sage: G3x3=graphs.MathonPseudocyclicStronglyRegularGraph(2,G=G,L=L)
sage: G3x3.is_strongly_regular(parameters=True)
(441, 220, 109, 110)
sage: G3x3.automorphism_group(algorithm="bliss").order() # optional - bliss
27
sage: G9=graphs.MathonPseudocyclicStronglyRegularGraph(2)
```

(continues on next page)

(continued from previous page)

```
sage: G9.is_strongly_regular(parameters=True)
(441, 220, 109, 110)
sage: G9.automorphism_group(algorithm="bliss").order() # optional - bliss
9
```

static MathonStronglyRegularGraph(*t*)

Return one of Mathon's graphs on 784 vertices.

INPUT:

- `t` (integer) – the number of the graph, from 0 to 2.

EXAMPLES:

```
sage: from sage.graphs.generators.smallgraphs import␣
↪MathonStronglyRegularGraph
sage: G = MathonStronglyRegularGraph(0)         # long time
sage: G.is_strongly_regular(parameters=True)    # long time
(784, 243, 82, 72)
```

static McGeeGraph(*embedding=2*)

Return the McGee Graph.

See the Wikipedia article McGee_graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.McGeeGraph()
sage: g.order()
24
sage: g.size()
36
sage: g.girth()
7
sage: g.diameter()
4
sage: g.show()
sage: graphs.McGeeGraph(embedding=1).show()
```

static McLaughlinGraph()

Return the McLaughlin Graph.

The McLaughlin Graph is the unique strongly regular graph of parameters $(275, 112, 30, 56)$.

For more information on the McLaughlin Graph, see its web page on Andries Brouwer's website which gives the definition that this method implements.

Note: To create this graph you must have the gap_packages spkg installed.

EXAMPLES:

```
sage: g = graphs.McLaughlinGraph()            # optional gap_packages
sage: g.is_strongly_regular(parameters=True) # optional gap_packages
(275, 112, 30, 56)
```

(continues on next page)

(continued from previous page)

```
sage: set(g.spectrum()) == {112, 2, -28}      # optional gap_packages
True
```

static MeredithGraph()

Return the Meredith Graph.

The Meredith Graph is a 4-regular 4-connected non-hamiltonian graph. For more information on the Meredith Graph, see the Wikipedia article Meredith_graph.

EXAMPLES:

```
sage: g = graphs.MeredithGraph()
sage: g.is_regular(4)
True
sage: g.order()
70
sage: g.size()
140
sage: g.radius()
7
sage: g.diameter()
8
sage: g.girth()
4
sage: g.chromatic_number()
3
sage: g.is_hamiltonian() # long time
False
```

static MoebiusKantorGraph()

Return a Möbius-Kantor Graph.

A Möbius-Kantor graph is a cubic symmetric graph. (See also the Heawood graph). It has 16 nodes and 24 edges. It is nonplanar and Hamiltonian. It has diameter = 4, girth = 6, and chromatic number = 2. It is identical to the Generalized Petersen graph, P[8,3].

For more details, see Möbius-Kantor Graph - from Wolfram MathWorld.

PLOTTING: See the plotting section for the generalized Petersen graphs.

EXAMPLES:

```
sage: MK = graphs.MoebiusKantorGraph()
sage: MK
Moebius-Kantor Graph: Graph on 16 vertices
sage: MK.graph6_string()
'OhCGKE?O@?ACAC@I?Q_AS'
sage: (graphs.MoebiusKantorGraph()).show() # long time
```

static MoserSpindle()

Return the Moser spindle.

For more information, see the Wikipedia article Moser_spindle.

EXAMPLES:

The Moser spindle is a planar graph having 7 vertices and 11 edges:

```
sage: G = graphs.MoserSpindle(); G
Moser spindle: Graph on 7 vertices
sage: G.is_planar()
True
sage: G.order()
7
sage: G.size()
11
```

It is a Hamiltonian graph with radius 2, diameter 2, and girth 3:

```
sage: G.is_hamiltonian()
True
sage: G.radius()
2
sage: G.diameter()
2
sage: G.girth()
3
```

The Moser spindle can be drawn in the plane as a unit distance graph, has chromatic number 4, and its automorphism group is isomorphic to the dihedral group D_4:

```
sage: pos = G.get_pos()
sage: all(sum((ui-vi)**2 for ui, vi in zip(pos[u], pos[v])) == 1
....:         for u, v in G.edge_iterator(labels=None))
True
sage: G.chromatic_number()
4
sage: ag = G.automorphism_group()
sage: ag.is_isomorphic(DihedralGroup(4))
True
```

static MuzychukS6Graph(*n*, *d*, *Phi='fixed'*, *Sigma='fixed'*, *verbose=False*)

Return a strongly regular graph of S6 type from [Muz2007] on $n^d((n^d-1)/(n-1)+1)$ vertices.

The construction depends upon a number of parameters, two of them, n and d, mandatory, and Φ and Σ mappings defined in [Muz2007]. These graphs have parameters $(mn^d, n^{d-1}(m-1)-1, \mu-2, \mu)$, where $\mu = \frac{n^{d-1}-1}{n-1}n^{d-1}$ and $m := \frac{n^d-1}{n-1}+1$.

Some details on Φ and Σ are as follows. Let L be the complete graph on $M := \{0, ..., m-1\}$ with the matching $\{(2i, 2i+1) | i = 0, ..., m/2\}$ removed. Then one arbitrarily chooses injections Φ_i from the edges of L on $i \in M$ into sets of parallel classes of affine d-dimensional designs; our implementation uses the designs of hyperplanes in d-dimensional affine geometries over $GF(n)$. Finally, for each edge ij of L one arbitrarily chooses bijections Σ_{ij} between Φ_i and Φ_j. More details, in particular how these choices lead to non-isomorphic graphs, are in [Muz2007].

INPUT:

- `n` (integer)– a prime power
- `d` (integer)– must be odd if n is odd
- `Phi` is an optional parameter of the construction; it must be either
  - 'fixed'– this will generate fixed default Φ_i, for $i \in M$, or
  - 'random'– Φ_i are generated at random, or

  - A dictionary describing the functions Φ_i; for $i \in M$, Phi[(i, T)] in M, for each edge T of L on i. Also, each Φ_i must be injective.

- `Sigma` is an optional parameter of the construction; it must be either
  - 'fixed'– this will generate a fixed default Σ, or
  - 'random'– Σ is generated at random.
- `verbose` (Boolean)– default is False. If True, print progress information

See also:

- *is_muzychuk_S6()*

Todo: Implement the possibility to explicitly supply the parameter Σ of the construction.

EXAMPLES:

```
sage: graphs.MuzychukS6Graph(3, 3).is_strongly_regular(parameters=True)
(378, 116, 34, 36)
sage: phi={(2,(0,2)):0,(1,(1,3)):1,(0,(0,3)):1,(2,(1,2)):1,(1,(1,
....:  2)):0,(0,(0,2)):0,(3,(0,3)):0,(3,(1,3)):1}
sage: graphs.MuzychukS6Graph(2,2,Phi=phi).is_strongly_regular(parameters=True)
(16, 5, 0, 2)
```

static MycielskiGraph(*k=1*, *relabel=True*)

Returns the k-th Mycielski Graph.

The graph M_k is triangle-free and has chromatic number equal to k. These graphs show, constructively, that there are triangle-free graphs with arbitrarily high chromatic number.

The Mycielski graphs are built recursively starting with M_0, an empty graph; M_1, a single vertex graph; and M_2 is the graph K_2. M_{k+1} is then built from M_k as follows:

If the vertices of M_k are $v_1, \ldots, v_n$, then the vertices of M_{k+1} are $v_1, \ldots, v_n, w_1, \ldots, w_n, z$. Vertices $v_1, \ldots, v_n$ induce a copy of M_k. Vertices $w_1, \ldots, w_n$ are an independent set. Vertex z is adjacent to all the w_i-vertices. Finally, vertex w_i is adjacent to vertex v_j iff v_i is adjacent to v_j.

INPUT:

- `k` Number of steps in the construction process.
- `relabel` Relabel the vertices so their names are the integers `range(n)` where `n` is the number of vertices in the graph.

EXAMPLES:

The Mycielski graph M_k is triangle-free and has chromatic number equal to k.

```
sage: g = graphs.MycielskiGraph(5)
sage: g.is_triangle_free()
True
sage: g.chromatic_number()
5
```

The graphs M_4 is (isomorphic to) the Grotzsch graph.

```
sage: g = graphs.MycielskiGraph(4)
sage: g.is_isomorphic(graphs.GrotzschGraph())
True
```

REFERENCES:

- [1] Weisstein, Eric W. "Mycielski Graph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MycielskiGraph.html

static MycielskiStep(*g*)

Perform one iteration of the Mycielski construction.

See the documentation for `MycielskiGraph` which uses this method. We expose it to all users in case they may find it useful.

EXAMPLE. One iteration of the Mycielski step applied to the 5-cycle yields a graph isomorphic to the Grotzsch graph

```
sage: g = graphs.CycleGraph(5)
sage: h = graphs.MycielskiStep(g)
sage: h.is_isomorphic(graphs.GrotzschGraph())
True
```

static NKStarGraph(*n*, *k*)

Returns the (n,k)-star graph.

The vertices of the (n,k)-star graph are the set of all arrangements of n symbols into labels of length k. There are two adjacency rules for the (n,k)-star graph. First, two vertices are adjacent if one can be obtained from the other by swapping the first symbol with another symbol. Second, two vertices are adjacent if one can be obtained from the other by swapping the first symbol with an external symbol (a symbol not used in the original label).

INPUT:

- `n`
- `k`

EXAMPLES:

```
sage: g = graphs.NKStarGraph(4,2)
sage: g.plot() # long time
Graphics object consisting of 31 graphics primitives
```

REFERENCES:

- Wei-Kuo, Chiang, and Chen Rong-Jaye. "The (n, k)-star graph: A generalized star graph." Information Processing Letters 56, no. 5 (December 8, 1995): 259-264.

AUTHORS:

- Michael Yurko (2009-09-01)

static NStarGraph(*n*)

Returns the n-star graph.

The vertices of the n-star graph are the set of permutations on n symbols. There is an edge between two vertices if their labels differ only in the first and one other position.

INPUT:

- `n`

EXAMPLES:

```
sage: g = graphs.NStarGraph(4)
sage: g.plot() # long time
Graphics object consisting of 61 graphics primitives
```

REFERENCES:

- S.B. Akers, D. Horel and B. Krishnamurthy, The star graph: An attractive alternative to the previous n-cube. In: Proc. Internat. Conf. on Parallel Processing (1987), pp. 393–400.

AUTHORS:

- Michael Yurko (2009-09-01)

static NauruGraph(*embedding=2*)
Return the Nauru Graph.

See the Wikipedia article Nauru_graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.NauruGraph()
sage: g.order()
24
sage: g.size()
36
sage: g.girth()
6
sage: g.diameter()
4
sage: g.show()
sage: graphs.NauruGraph(embedding=1).show()
```

static NonisotropicOrthogonalPolarGraph(*m*, *q*, *sign='+'*, *perp=None*)
Returns the Graph $NO^{\epsilon,\perp}_m(q)$

Let the vectorspace of dimension m over F_q be endowed with a nondegenerate quadratic form F, of type `sign` for m even.

- m even: assume further that $q = 2$ or 3. Returns the graph of the points (in the underlying projective space) x satisfying $F(x) = 1$, with adjacency given by orthogonality w.r.t. F. Parameter `perp` is ignored.
- m odd: if `perp` is not `None`, then we assume that $q = 5$ and return the graph of the points x satisfying $F(x) = \pm 1$ if `sign="+"`, respectively $F(x) \in \{2, 3\}$ if `sign="-"`, with adjacency given by orthogonality w.r.t. F (cf. Sect 7.D of [BL1984]). Otherwise return the graph of nongenerate hyperplanes of type `sign`, adjacent whenever the intersection is degenerate (cf. Sect. 7.C of [BL1984]). Note that for $q = 2$ one will get a complete graph.

For more information, see Sect. 9.9 of [**?**] and [BL1984]. Note that the page of Andries Brouwer's website uses different notation.

INPUT:

- `m` - integer, half the dimension of the underlying vectorspace
- `q` - a power of a prime number, the size of the underlying field

- `sign` – `"+"` (default) or `"-"`.

EXAMPLES:

$NO^-(4,2)$ is isomorphic to Petersen graph:

```
sage: g=graphs.NonisotropicOrthogonalPolarGraph(4,2,'-'); g
NO^-(4, 2): Graph on 10 vertices
sage: g.is_strongly_regular(parameters=True)
(10, 3, 0, 1)
```

$NO^-(6,2)$ and $NO^+(6,2)$:

```
sage: g=graphs.NonisotropicOrthogonalPolarGraph(6,2,'-')
sage: g.is_strongly_regular(parameters=True)
(36, 15, 6, 6)
sage: g=graphs.NonisotropicOrthogonalPolarGraph(6,2,'+'); g
NO^+(6, 2): Graph on 28 vertices
sage: g.is_strongly_regular(parameters=True)
(28, 15, 6, 10)
```

$NO^+(8,2)$:

```
sage: g=graphs.NonisotropicOrthogonalPolarGraph(8,2,'+')
sage: g.is_strongly_regular(parameters=True)
(120, 63, 30, 36)
```

Wilbrink's graphs for $q = 5$:

```
sage: graphs.NonisotropicOrthogonalPolarGraph(5,5,perp=1).is_strongly_
↪regular(parameters=True) # long time
(325, 60, 15, 10)
sage: graphs.NonisotropicOrthogonalPolarGraph(5,5,'-',perp=1).is_strongly_
↪regular(parameters=True) # long time
(300, 65, 10, 15)
```

Wilbrink's graphs:

```
sage: g=graphs.NonisotropicOrthogonalPolarGraph(5,4,'+')
sage: g.is_strongly_regular(parameters=True)
(136, 75, 42, 40)
sage: g=graphs.NonisotropicOrthogonalPolarGraph(5,4,'-')
sage: g.is_strongly_regular(parameters=True)
(120, 51, 18, 24)
sage: g=graphs.NonisotropicOrthogonalPolarGraph(7,4,'+'); g # not tested␣
↪(long time)
NO^+(7, 4): Graph on 2080 vertices
sage: g.is_strongly_regular(parameters=True) # not tested (long time)
(2080, 1071, 558, 544)
```

static NonisotropicUnitaryPolarGraph(m, q)

Returns the Graph $NU(m,q)$.

Returns the graph on nonisotropic, with respect to a nondegenerate Hermitean form, points of the $(m-1)$-dimensional projective space over F_q, with points adjacent whenever they lie on a tangent (to the set of isotropic points) line. For more information, see Sect. 9.9 of [**?**] and series C14 in [Hub1975].

INPUT:

- `m, q` (integers) – q must be a prime power.

EXAMPLES:

```
sage: g=graphs.NonisotropicUnitaryPolarGraph(5,2); g
NU(5, 2): Graph on 176 vertices
sage: g.is_strongly_regular(parameters=True)
(176, 135, 102, 108)
```

static Nowhere0WordsTwoWeightCodeGraph(*q*, *hyperoval=None*, *field=None*, *check_hyperoval=True*)

Return the subgraph of nowhere 0 words from two-weight code of projective plane hyperoval.

Let $q = 2^k$ and $\Pi = PG(2, q)$. Fix a hyperoval $O \subset \Pi$. Let $V = F_q^3$ and C the two-weight 3-dimensional linear code over F_q with words $c(v)$ obtained from $v \in V$ by computing

$$c(v) = (\langle v, o_1 \rangle, ..., \langle v, o_{q+2} \rangle), o_j \in O.$$

C contains $q(q-1)^2/2$ words without 0 entries. The subgraph of the strongly regular graph of C induced on the latter words is also strongly regular, assuming $q > 4$. This is a construction due to A.E.Brouwer [Bro2016], and leads to graphs with parameters also given by a construction in [HHL2009]. According to [Bro2016], these two constructions are likely to produce isomorphic graphs.

INPUT:

- `q` – a power of two
- `hyperoval` – a hyperoval (i.e. a complete 2-arc; a set of points in the plane meeting every line in 0 or 2 points) in $PG(2, q)$ over the field `field`. Each point of `hyperoval` must be a length 3 vector over `field` with 1st non-0 coordinate equal to 1. By default, `hyperoval` and `field` are not specified, and constructed on the fly. In particular, `hyperoval` we build is the classical one, i.e. a conic with the point of intersection of its tangent lines.
- `field` – an instance of a finite field of order q, must be provided if `hyperoval` is provided.
- `check_hyperoval` – (default: `True`) if `True`, check `hyperoval` for correctness.

See also:

- *is_nowhere0_twoweight()*

EXAMPLES:

using the built-in construction:

```
sage: g=graphs.Nowhere0WordsTwoWeightCodeGraph(8); g
Nowhere0WordsTwoWeightCodeGraph(8): Graph on 196 vertices
sage: g.is_strongly_regular(parameters=True)
(196, 60, 14, 20)
sage: g=graphs.Nowhere0WordsTwoWeightCodeGraph(16) # not tested (long time)
sage: g.is_strongly_regular(parameters=True)       # not tested (long time)
(1800, 728, 268, 312)
```

supplying your own hyperoval:

```
sage: F=GF(8)
sage: O=[vector(F,(0,0,1)),vector(F,(0,1,0))]+[vector(F, (1,x^2,x)) for x in␣
↪F]
sage: g=graphs.Nowhere0WordsTwoWeightCodeGraph(8,hyperoval=O,field=F); g
Nowhere0WordsTwoWeightCodeGraph(8): Graph on 196 vertices
sage: g.is_strongly_regular(parameters=True)
(196, 60, 14, 20)
```

static OctahedralGraph()

Returns an Octahedral graph (with 6 nodes).

The regular octahedron is an 8-sided polyhedron with triangular faces. The octahedral graph corresponds to the connectivity of the vertices of the octahedron. It is the line graph of the tetrahedral graph. The octahedral is symmetric, so the spring-layout algorithm will be very effective for display.

PLOTTING: The Octahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show an Octahedral graph

```
sage: g = graphs.OctahedralGraph()
sage: g.show() # long time
```

Create several octahedral graphs in a Sage graphics array They will be drawn differently due to the use of the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.OctahedralGraph()
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static OddGraph(n)

Returns the Odd Graph with parameter n.

The Odd Graph with parameter n is defined as the Kneser Graph with parameters $2n-1, n-1$. Equivalently, the Odd Graph is the graph whose vertices are the $n-1$-subsets of $[0, 1, \ldots, 2(n-1)]$, and such that two vertices are adjacent if their corresponding sets are disjoint.

For example, the Petersen Graph can be defined as the Odd Graph with parameter 3.

EXAMPLES:

```
sage: OG = graphs.OddGraph(3)
sage: sorted(OG.vertex_iterator(), key=str)
[{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5},
 {3, 4}, {3, 5}, {4, 5}]
sage: P = graphs.PetersenGraph()
sage: P.is_isomorphic(OG)
True
```

static OrthogonalArrayBlockGraph(k, n, *OA=None*)

Return the graph of an $OA(k, n)$.

The intersection graph of the blocks of a transversal design with parameters (k, n), or $TD(k, n)$ for short, is a strongly regular graph (unless it is a complete graph). Its parameters (v, k', λ, μ) are determined by the parameters k, n via:

$$v = n^2, k' = k(n-1), \lambda = (k-1)(k-2) + n - 2, \mu = k(k-1)$$

As transversal designs and orthogonal arrays (OA for short) are equivalent objects, this graph can also be built from the blocks of an $OA(k,n)$, two of them being adjacent if one of their coordinates match.

For more information on these graphs, see Andries Brouwer's page on Orthogonal Array graphs.

Warning:

- Brouwer's website uses the notation $OA(n,k)$ instead of $OA(k,n)$
- For given parameters k and n there can be many $OA(k,n)$: the graphs returned are not uniquely defined by their parameters (see the examples below).
- If the function is called only with the parameter `k` and `n` the results might be different with two versions of Sage, or even worse : some could not be available anymore.

See also:

`sage.combinat.designs.orthogonal_arrays`

INPUT:

- `k,n` (integers)
- `OA` – An orthogonal array. If set to `None` (default) then `orthogonal_array()` is called to compute an $OA(k,n)$.

EXAMPLES:

```
sage: G = graphs.OrthogonalArrayBlockGraph(5,5); G
OA(5,5): Graph on 25 vertices
sage: G.is_strongly_regular(parameters=True)
(25, 20, 15, 20)
sage: G = graphs.OrthogonalArrayBlockGraph(4,10); G
OA(4,10): Graph on 100 vertices
sage: G.is_strongly_regular(parameters=True)
(100, 36, 14, 12)
```

Two graphs built from different orthogonal arrays are also different:

```
sage: k=4;n=10
sage: OAa = designs.orthogonal_arrays.build(k,n)
sage: OAb = [[(x+1)%n for x in R] for R in OAa]
sage: set(map(tuple,OAa)) == set(map(tuple,OAb))
False
sage: Ga = graphs.OrthogonalArrayBlockGraph(k,n,OAa)
sage: Gb = graphs.OrthogonalArrayBlockGraph(k,n,OAb)
sage: Ga == Gb
False
```

As `OAb` was obtained from `OAa` by a relabelling the two graphs are isomorphic:

```
sage: Ga.is_isomorphic(Gb)
True
```

But there are examples of $OA(k,n)$ for which the resulting graphs are not isomorphic:

```
sage: oa0 = [[0, 0, 1], [0, 1, 3], [0, 2, 0], [0, 3, 2],
....:        [1, 0, 3], [1, 1, 1], [1, 2, 2], [1, 3, 0],
....:        [2, 0, 0], [2, 1, 2], [2, 2, 1], [2, 3, 3],
```

(continues on next page)

(continued from previous page)

```
....:         [3, 0, 2], [3, 1, 0], [3, 2, 3], [3, 3, 1]]
sage: oa1 = [[0, 0, 1], [0, 1, 0], [0, 2, 3], [0, 3, 2],
....:         [1, 0, 3], [1, 1, 2], [1, 2, 0], [1, 3, 1],
....:         [2, 0, 0], [2, 1, 1], [2, 2, 2], [2, 3, 3],
....:         [3, 0, 2], [3, 1, 3], [3, 2, 1], [3, 3, 0]]
sage: g0 = graphs.OrthogonalArrayBlockGraph(3,4,oa0)
sage: g1 = graphs.OrthogonalArrayBlockGraph(3,4,oa1)
sage: g0.is_isomorphic(g1)
False
```

But nevertheless isospectral:

```
sage: g0.spectrum()
[9, 1, 1, 1, 1, 1, 1, 1, 1, 1, -3, -3, -3, -3, -3, -3]
sage: g1.spectrum()
[9, 1, 1, 1, 1, 1, 1, 1, 1, 1, -3, -3, -3, -3, -3, -3]
```

Note that the graph `g0` is actually isomorphic to the affine polar graph $VO^+(4,2)$:

```
sage: graphs.AffineOrthogonalPolarGraph(4,2,'+').is_isomorphic(g0)
True
```

static OrthogonalPolarGraph(m, q, *sign='+'*)

Returns the Orthogonal Polar Graph $O^\epsilon(m,q)$.

For more information on Orthogonal Polar graphs, see the page of Andries Brouwer's website.

INPUT:

- `m`, `q` (integers) – q must be a prime power.
- `sign` – `"+"` or `"-"` if m is even, `"+"` (default) otherwise.

EXAMPLES:

```
sage: G = graphs.OrthogonalPolarGraph(6,3,"+"); G
Orthogonal Polar Graph O^+(6, 3): Graph on 130 vertices
sage: G.is_strongly_regular(parameters=True)
(130, 48, 20, 16)
sage: G = graphs.OrthogonalPolarGraph(6,3,"-"); G
Orthogonal Polar Graph O^-(6, 3): Graph on 112 vertices
sage: G.is_strongly_regular(parameters=True)
(112, 30, 2, 10)
sage: G = graphs.OrthogonalPolarGraph(5,3); G
Orthogonal Polar Graph O(5, 3): Graph on 40 vertices
sage: G.is_strongly_regular(parameters=True)
(40, 12, 2, 4)
sage: G = graphs.OrthogonalPolarGraph(8,2,"+"); G
Orthogonal Polar Graph O^+(8, 2): Graph on 135 vertices
sage: G.is_strongly_regular(parameters=True)
(135, 70, 37, 35)
sage: G = graphs.OrthogonalPolarGraph(8,2,"-"); G
Orthogonal Polar Graph O^-(8, 2): Graph on 119 vertices
sage: G.is_strongly_regular(parameters=True)
(119, 54, 21, 27)
```

static PaleyGraph(q)

Paley graph with q vertices

Parameter q must be the power of a prime number and congruent to 1 mod 4.

EXAMPLES:

```
sage: G = graphs.PaleyGraph(9); G
Paley graph with parameter 9: Graph on 9 vertices
sage: G.is_regular()
True
```

A Paley graph is always self-complementary:

```
sage: G.is_self_complementary()
True
```

static PappusGraph()

Return the Pappus graph, a graph on 18 vertices.

The Pappus graph is cubic, symmetric, and distance-regular.

EXAMPLES:

```
sage: G = graphs.PappusGraph()
sage: G.show()  # long time
sage: L = graphs.LCFGraph(18, [5,7,-7,7,-7,-5], 3)
sage: L.show()  # long time
sage: G.is_isomorphic(L)
True
```

static PasechnikGraph(n)

Pasechnik strongly regular graph on $(4n-1)^2$ vertices

A strongly regular graph with parameters of the orthogonal array graph *OrthogonalArrayBlockGraph()*, also known as pseudo Latin squares graph $L_{2n-1}(4n-1)$, constructed from a skew Hadamard matrix of order $4n$ following [Pas1992].

See also:

- *is_orthogonal_array_block_graph()*

EXAMPLES:

```
sage: graphs.PasechnikGraph(4).is_strongly_regular(parameters=True)
(225, 98, 43, 42)
sage: graphs.PasechnikGraph(5).is_strongly_regular(parameters=True)  # long␣
↪time
(361, 162, 73, 72)
sage: graphs.PasechnikGraph(9).is_strongly_regular(parameters=True)  # not␣
↪tested
(1225, 578, 273, 272)
```

static PathGraph(n, *pos=None*)

Return a path graph with n nodes.

A path graph is a graph where all inner nodes are connected to their two neighbors and the two end-nodes are connected to their one inner neighbors (i.e.: a cycle graph without the first and last node connected).

INPUT:

- `n` – number of nodes of the path graph

- `pos` (default: `None`) – a string which is either 'circle' or 'line' (otherwise the default is used) indicating which embedding algorithm to use. See the plotting section below for more detail.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the graph may be drawn in one of two ways: The 'line' argument will draw the graph in a horizontal line (left to right) if there are less than 11 nodes. Otherwise the 'line' argument will append horizontal lines of length 10 nodes below, alternating left to right and right to left. The 'circle' argument will cause the graph to be drawn in a cycle-shape, with the first node at the top and then about the circle in a clockwise manner. By default (without an appropriate string argument) the graph will be drawn as a 'circle' if $10 < n < 41$ and as a 'line' for all other n.

EXAMPLES: Show default drawing by size: 'line': $n \leq 10$

```
sage: p = graphs.PathGraph(10)
sage: p.show() # long time
```

'circle': $10 < n < 41$

```
sage: q = graphs.PathGraph(25)
sage: q.show() # long time
```

'line': $n \geq 41$

```
sage: r = graphs.PathGraph(55)
sage: r.show() # long time
```

Override the default drawing:

```
sage: s = graphs.PathGraph(5,'circle')
sage: s.show() # long time
```

static PerkelGraph()

Return the Perkel Graph.

The Perkel Graph is a 6-regular graph with 57 vertices and 171 edges. It is the unique distance-regular graph with intersection array $(6, 5, 2; 1, 1, 3)$. For more information, see the Wikipedia article Perkel_graph or https://www.win.tue.nl/~aeb/graphs/Perkel.html.

EXAMPLES:

```
sage: g = graphs.PerkelGraph(); g
Perkel Graph: Graph on 57 vertices
sage: g.is_distance_regular(parameters=True)
([6, 5, 2, None], [None, 1, 1, 3])
```

static PermutationGraph(*second_permutation*, *first_permutation=None*)

Build a permutation graph from one permutation or from two lists.

Definition:

If σ is a permutation of $\{1, 2, \ldots, n\}$, then the permutation graph of σ is the graph on vertex set $\{1, 2, \ldots, n\}$ in which two vertices i and j satisfying $i < j$ are connected by an edge if and only if $\sigma^{-1}(i) > \sigma^{-1}(j)$. A visual way to construct this graph is as follows:

> Take two horizontal lines in the euclidean plane, and mark points $1, ..., n$ from left to right on the first of them. On the second one, still from left to right, mark n points $\sigma(1), \sigma(2), \ldots, \sigma(n)$. Now, link by a segment the two points marked with 1, then link together the points marked with 2, and so on. The permutation graph of σ is the intersection graph of those segments: there exists a vertex in this graph for each element from 1 to n, two vertices i, j being adjacent if the segments i and j cross each other.

The set of edges of the permutation graph can thus be identified with the set of inversions of the inverse of the given permutation σ.

A more general notion of permutation graph can be defined as follows: If S is a set, and $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ are two lists of elements of S, each of which lists contains every element of S exactly once, then the permutation graph defined by these two lists is the graph on the vertex set S in which two vertices i and j are connected by an edge if and only if the order in which these vertices appear in the list $(a_1, a_2, \ldots, a_n)$ is the opposite of the order in which they appear in the list $(b_1, b_2, \ldots, b_n)$. When $(a_1, a_2, \ldots, a_n) = (1, 2, \ldots, n)$, this graph is the permutation graph of the permutation $(b_1, b_2, \ldots, b_n) \in S_n$. Notice that S does not have to be a set of integers here, but can be a set of strings, tuples, or anything else. We can still use the above visual description to construct the permutation graph, but now we have to mark points $a_1, a_2, \ldots, a_n$ from left to right on the first horizontal line and points $b_1, b_2, \ldots, b_n$ from left to right on the second horizontal line.

INPUT:

- `second_permutation` – the unique permutation/list defining the graph, or the second of the two (if the graph is to be built from two permutations/lists).
- `first_permutation` (optional) – the first of the two permutations/lists from which the graph should be built, if it is to be built from two permutations/lists.

  When `first_permutation is None` (default), it is set to be equal to `sorted(second_permutation)`, which yields the expected ordering when the elements of the graph are integers.

See also:

- Recognition of Permutation graphs in the *comparability module*.
- Drawings of permutation graphs as intersection graphs of segments is possible through the `show()` method of `Permutation` objects.

  The correct argument to use in this case is `show(representation = "braid")`.
- `inversions()`

EXAMPLES:

```
sage: p = Permutations(5).random_element()
sage: PG = graphs.PermutationGraph(p)
sage: edges = PG.edges(labels=False)
sage: set(edges) == set(p.inverse().inversions())
True

sage: PG = graphs.PermutationGraph([3,4,5,1,2])
sage: sorted(PG.edges())
[(1, 3, None),
 (1, 4, None),
 (1, 5, None),
 (2, 3, None),
 (2, 4, None),
 (2, 5, None)]
sage: PG = graphs.PermutationGraph([3,4,5,1,2], [1,4,2,5,3])
sage: sorted(PG.edges())
[(1, 3, None),
 (1, 4, None),
 (1, 5, None),
 (2, 3, None),
 (2, 5, None),
```

(continues on next page)

(continued from previous page)

```
 (3, 4, None),
 (3, 5, None)]
sage: PG = graphs.PermutationGraph([1,4,2,5,3], [3,4,5,1,2])
sage: sorted(PG.edges())
[(1, 3, None),
 (1, 4, None),
 (1, 5, None),
 (2, 3, None),
 (2, 5, None),
 (3, 4, None),
 (3, 5, None)]

sage: PG = graphs.PermutationGraph(Permutation([1,3,2]), Permutation([1,2,3]))
sage: sorted(PG.edges())
[(2, 3, None)]

sage: graphs.PermutationGraph([]).edges()
[]
sage: graphs.PermutationGraph([], []).edges()
[]

sage: PG = graphs.PermutationGraph("graph", "phrag")
sage: sorted(PG.edges())
[('a', 'g', None),
 ('a', 'h', None),
 ('a', 'p', None),
 ('g', 'h', None),
 ('g', 'p', None),
 ('g', 'r', None),
 ('h', 'r', None),
 ('p', 'r', None)]
```

static PetersenGraph()

Return the Petersen Graph.

The Petersen Graph is a named graph that consists of 10 vertices and 15 edges, usually drawn as a five-point star embedded in a pentagon.

The Petersen Graph is a common counterexample. For example, it is not Hamiltonian.

PLOTTING: See the plotting section for the generalized Petersen graphs.

EXAMPLES: We compare below the Petersen graph with the default spring-layout versus a planned position dictionary of [x,y] tuples:

```
sage: petersen_spring = Graph({0:[1,4,5], 1:[0,2,6], 2:[1,3,7], 3:[2,4,8],␣
↪4:[0,3,9], 5:[0,7,8], 6:[1,8,9], 7:[2,5,9], 8:[3,5,6], 9:[4,6,7]})
sage: petersen_spring.show() # long time
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

static PoussinGraph()

Return the Poussin Graph.

For more information on the Poussin Graph, see its corresponding Wolfram page.

EXAMPLES:

```
sage: g = graphs.PoussinGraph()
sage: g.order()
15
sage: g.is_planar()
True
```

static QueenGraph(*dim_list*, *radius=None*, *relabel=False*)

Returns the d-dimensional Queen Graph with prescribed dimensions.

The 2-dimensional Queen Graph of parameters n and m is a graph with nm vertices in which each vertex represents a square in an $n \times m$ chessboard, and each edge corresponds to a legal move by a queen.

The d-dimensional Queen Graph with $d >= 2$ has for vertex set the cells of a d-dimensional grid with prescribed dimensions, and each edge corresponds to a legal move by a queen in either one or two dimensions.

All 2-dimensional Queen Graphs are Hamiltonian and biconnected. The chromatic number of a (n, n)-Queen Graph is at least n, and it is exactly n when $n \equiv 1, 5 \bmod 6$.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `radius` – (default: `None`) by setting the radius to a positive integer, one may reduce the visibility of the queen to at most `radius` steps. When radius is 1, the resulting graph is a King Graph.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

EXAMPLES:

The $(2, 2)$-Queen Graph is isomorphic to the complete graph on 4 vertices:

```
sage: G = graphs.QueenGraph( [2, 2] )
sage: G.is_isomorphic( graphs.CompleteGraph(4) )
True
```

The Queen Graph with radius 1 is isomorphic to the King Graph:

```
sage: G = graphs.QueenGraph( [4, 5], radius=1 )
sage: H = graphs.KingGraph( [5, 4] )
sage: G.is_isomorphic( H )
True
```

Also True in higher dimensions:

```
sage: G = graphs.QueenGraph( [3, 4, 5], radius=1 )
sage: H = graphs.KingGraph( [5, 3, 4] )
sage: G.is_isomorphic( H )
True
```

The Queen Graph can be obtained from the Rook Graph and the Bishop Graph:

```
sage: for d in range(3,12):   # long time
....:     for r in range(1,d+1):
....:         G = graphs.QueenGraph([d,d],radius=r)
....:         H = graphs.RookGraph([d,d],radius=r)
....:         B = graphs.BishopGraph([d,d],radius=r)
....:         H.add_edges(B.edges())
```

(continues on next page)

(continued from previous page)

```
....:         if not G.is_isomorphic(H):
....:             print("that's not good!")
```

static RandomBarabasiAlbert(*n*, *m*, *seed=None*)

Return a random graph created using the Barabasi-Albert preferential attachment model.

A graph with m vertices and no edges is initialized, and a graph of n vertices is grown by attaching new vertices each with m edges that are attached to existing vertices, preferentially with high degree.

INPUT:

- `n` – number of vertices in the graph
- `m` – number of edges to attach from each new node
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

EXAMPLES:

We show the edge list of a random graph on 6 nodes with $m = 2$:

```
sage: G = graphs.RandomBarabasiAlbert(6,2)
sage: G.order(), G.size()
(6, 8)
sage: G.degree_sequence()  # random
[4, 3, 3, 2, 2, 2]
```

We plot a random graph on 12 nodes with $m = 3$:

```
sage: ba = graphs.RandomBarabasiAlbert(12,3)
sage: ba.show()  # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(1,10):
....:     k = graphs.RandomBarabasiAlbert(i+3, 3)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show()  # long time
```

When $m = 1$, the generated graph is a tree:

```
sage: graphs.RandomBarabasiAlbert(6, 1).is_tree()
True
```

static RandomBicubicPlanar(*n*)

Return the graph of a random bipartite cubic map with $3n$ edges.

INPUT:

n – an integer (at least 1)

OUTPUT:

a graph with multiple edges (no embedding is provided)

The algorithm used is described in [Sch1999]. This samples a random rooted bipartite cubic map, chosen uniformly at random.

First one creates a random binary tree with n vertices. Next one turns this into a blossoming tree (at random) and reads the contour word of this blossoming tree.

Then one performs a rotation on this word so that this becomes a balanced word. There are three ways to do that, one is picked at random. Then a graph is build from the balanced word by iterated closure (adding edges).

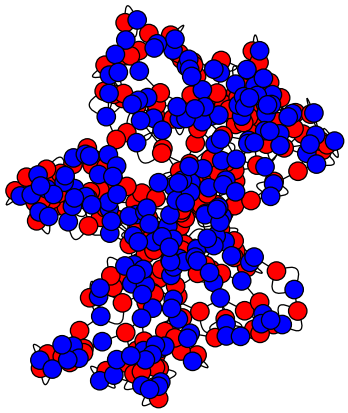
In the returned graph, the three edges incident to any given vertex are colored by the integers 0, 1 and 2.

See also:

the auxiliary method `blossoming_contour()`

EXAMPLES:

```
sage: n = randint(200, 300)
sage: G = graphs.RandomBicubicPlanar(n)
sage: G.order() == 2*n
True
sage: G.size() == 3*n
True
sage: G.is_bipartite() and G.is_planar() and G.is_regular(3)
True
sage: dic = {'red':[v for v in G.vertices() if v[0] == 'n'],
....:        'blue': [v for v in G.vertices() if v[0] != 'n']}
sage: G.plot(vertex_labels=False,vertex_size=20,vertex_colors=dic)
Graphics object consisting of ... graphics primitives
```



static RandomBipartite(*n1*, *n2*, *p*, *set_position=False*)

Returns a bipartite graph with $n1+n2$ vertices such that any edge from $[n1]$ to $[n2]$ exists with probability p.

INPUT:

- `n1, n2` – Cardinalities of the two sets
- `p` – Probability for an edge to exist
- `set_position` – boolean (default `False`); if set to `True`, we assign positions to the vertices so that the set of cardinality $n1$ is on the line $y = 1$ and the set of cardinality $n2$ is on the line $y = 0$.

EXAMPLES:

```
sage: g = graphs.RandomBipartite(5, 2, 0.5)
sage: g.vertices()
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1)]
```

static RandomBlockGraph(*m*, *k*, *kmax=None*, *incidence_structure=False*)

Return a Random Block Graph.

A block graph is a connected graph in which every biconnected component (block) is a clique.

See also:

- Wikipedia article Block_graph for more details on these graphs
- *is_block_graph()* – test if a graph is a block graph
- *blocks_and_cut_vertices()*
- *blocks_and_cuts_tree()*
- *IncidenceStructure()*

INPUT:

- `m` – integer; number of blocks (at least one).
- `k` – integer; minimum number of vertices of a block (at least two).
- `kmax` – integer (default: `None`) By default, each block has k vertices. When the parameter $kmax$ is specified (with $kmax \geq k$), the number of vertices of each block is randomly chosen between k and $kmax$.
- `incidence_structure` – boolean (default: `False`) when set to `True`, the incidence structure of the graphs is returned instead of the graph itself, that is the list of the lists of vertices in each block. This is useful for the creation of some hypergraphs.

OUTPUT:

A Graph when `incidence_structure==False` (default), and otherwise an incidence structure.

EXAMPLES:

A block graph with a single block is a clique:

```
sage: B = graphs.RandomBlockGraph(1, 4)
sage: B.is_clique()
True
```

A block graph with blocks of order 2 is a tree:

```
sage: B = graphs.RandomBlockGraph(10, 2)
sage: B.is_tree()
True
```

Every biconnected component of a block graph is a clique:

```
sage: B = graphs.RandomBlockGraph(5, 3, kmax=6)
sage: blocks,cuts = B.blocks_and_cut_vertices()
sage: all(B.is_clique(block) for block in blocks)
True
```

A block graph with blocks of order k has $m * (k - 1) + 1$ vertices:

```
sage: m, k = 6, 4
sage: B = graphs.RandomBlockGraph(m, k)
sage: B.order() == m*(k-1)+1
True
```

Test recognition methods:

```
sage: B = graphs.RandomBlockGraph(6, 2, kmax=6)
sage: B.is_block_graph()
True
sage: B in graph_classes.Block
True
```

Asking for the incidence structure:

```
sage: m, k = 6, 4
sage: IS = graphs.RandomBlockGraph(m, k, incidence_structure=True)
sage: from sage.combinat.designs.incidence_structures import␣
↪IncidenceStructure
sage: IncidenceStructure(IS)
Incidence structure with 19 points and 6 blocks
sage: m*(k-1)+1
19
```

static RandomBoundedToleranceGraph(*n*)

Returns a random bounded tolerance graph.

The random tolerance graph is built from a random bounded tolerance representation by using the function $ToleranceGraph$. This representation is a list $((l_0, r_0, t_0), (l_1, r_1, t_1), ..., (l_k, r_k, t_k))$ where $k = n - 1$ and $I_i = (l_i, r_i)$ denotes a random interval and t_i a random positive value less then or equal to the length of the interval I_i. The width of the representation is limited to n**2 * 2**n.

Note: The tolerance representation used to create the graph can be recovered using `get_vertex()` or `get_vertices()`.

INPUT:

- `n` – number of vertices of the random graph.

EXAMPLES:

Every (bounded) tolerance graph is perfect. Hence, the chromatic number is equal to the clique number

```
sage: g = graphs.RandomBoundedToleranceGraph(8)
sage: g.clique_number() == g.chromatic_number()
True
```

static RandomChordalGraph(*n*, *algorithm='growing'*, *k=None*, *l=None*, *f=None*, *s=None*)

Return a random chordal graph of order `n`.

A Graph G is said to be chordal if it contains no induced hole (a cycle of length at least 4). Equivalently, G is chordal if it has a perfect elimination orderings, if each minimal separator is a clique, or if it is the intersection graphs of subtrees of a tree. See the Wikipedia article Chordal_graph.

This generator implements the algorithms proposed in [SHET2018] for generating random chordal graphs as the intersection graph of n subtrees of a tree of order n.

The returned graph is not necessarily connected.

INPUT:

- `n` – integer; the number of nodes of the graph
- `algorithm` – string (default: `"growing"`); the choice of the algorithm for randomly selecting n subtrees of a random tree of order n. Possible choices are:
  - `"growing"` – for each subtree T_i, the algorithm picks a size k_i randomly from $[1, k]$. Then a random node of T is chosen as the first node of T_i. In each of the subsequent $k_i - 1$ iterations, it picks a random node in the neighborhood of T_i and adds it to T_i.
  - `"connecting"` – for each subtree T_i, it first selects k_i nodes of T, where k_i is a random integer from a Poisson distribution with mean l. T_i is then generated to be the minimal subtree containing the selected k_i nodes. This implies that a subtree will most likely have many more nodes than those selected initially, and this must be taken into consideration when choosing l.
  - `"pruned"` – for each subtree T_i, it randomly selects a fraction f of the edges on the tree and removes them. The number of edges to delete, say l, is calculated as $\lfloor (n-1)f \rfloor$, which will leave $l + 1$ subtrees in total. Then, it determines the sizes of the $l + 1$ subtrees and stores the distinct values. Finally, it picks a random size k_i from the set of largest $100(1 - s)\%$ of distinct values, and randomly chooses a subtree with size k_i.
- `k` – integer (default: `None`); maximum size of a subtree. If not specified (`None`), the maximum size is set to $\sqrt{n}$. This parameter is used only when `algorithm="growing"`. See `growing_subtrees()` for more details.
- `l` – a strictly positive real number (default: `None`); mean of a Poisson distribution. If not specified, the mean in set to $\log_2 n$. This parameter is used only when `algorithm="connecting"`. See `connecting_nodes()` for more details.
- `f` – a rational number (default: `None`); the edge deletion fraction. This value must be choosen in $[0..1]$. If not specified, this parameter is set to $\frac{1}{n-1}$. This parameter is used only when `algorithm="pruned"`. See `pruned_tree()` for more details.
- `s` – a real number between 0 and 1 (default: `None`); selection barrier for the size of trees. If not specified, this parameter is set to 0.5. This parameter is used only when `algorithm="pruned"`. See `pruned_tree()` for more details.

EXAMPLES:

```
sage: from sage.graphs.generators.random import RandomChordalGraph
sage: T = RandomChordalGraph(20, algorithm="growing", k=5)
sage: T.is_chordal()
True
sage: T = RandomChordalGraph(20, algorithm="connecting", l=3)
sage: T.is_chordal()
True
sage: T = RandomChordalGraph(20, algorithm="pruned", f=1/3, s=.5)
sage: T.is_chordal()
True
```

See also:

- `growing_subtrees()`
- `connecting_nodes()`
- `pruned_tree()`
- Wikipedia article Chordal_graph
- *`is_chordal()`*

- *IntersectionGraph()*

static RandomGNM(*n*, *m*, *dense=False*, *seed=None*)

Returns a graph randomly picked out of all graphs on n vertices with m edges.

INPUT:

- `n` - number of vertices.
- `m` - number of edges.
- `dense` - whether to use NetworkX's dense_gnm_random_graph or gnm_random_graph
- `seed` - a `random.Random` seed or a Python `int` for the random number generator (default: `None`).

EXAMPLES: We show the edge list of a random graph on 5 nodes with 10 edges.

```
sage: graphs.RandomGNM(5, 10).edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3,␣
↪4)]
```

We plot a random graph on 12 nodes with m = 12.

```
sage: gnm = graphs.RandomGNM(12, 12)
sage: gnm.show()  # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.RandomGNM(i+3, i^2-i)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show()  # long time
```

static RandomGNP(*n*, *p*, *seed=None*, *fast=True*, *algorithm='Sage'*)

Returns a random graph on n nodes. Each edge is inserted independently with probability p.

INPUT:

- `n` – number of nodes of the graph
- `p` – probability of an edge
- `seed` - a `random.Random` seed or a Python `int` for the random number generator (default: `None`).
- `fast` – boolean set to True (default) to use the algorithm with time complexity in $O(n+m)$ proposed in [BB2005a]. It is designed for generating large sparse graphs. It is faster than other algorithms for *LARGE* instances (try it to know whether it is useful for you).
- `algorithm` – By default (`algorithm='Sage'`), this function uses the algorithm implemented in `sage.graphs.graph_generators_pyx.pyx`. When `algorithm='networkx'`, this function calls the NetworkX function `fast_gnp_random_graph`, unless `fast=False`, then `gnp_random_graph`. Try them to know which algorithm is the best for you. The `fast` parameter is not taken into account by the 'Sage' algorithm so far.

REFERENCES:

- [ER1959]
- [Gil1959]

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES: We show the edge list of a random graph on 6 nodes with probability $p = .4$:

```
sage: set_random_seed(0)
sage: graphs.RandomGNP(6, .4).edges(labels=False)
[(0, 1), (0, 5), (1, 2), (2, 4), (3, 4), (3, 5), (4, 5)]
```

We plot a random graph on 12 nodes with probability $p = .71$:

```
sage: gnp = graphs.RandomGNP(12,.71)
sage: gnp.show() # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.RandomGNP(i+3,.43)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
sage: graphs.RandomGNP(4,1)
Complete graph: Graph on 4 vertices
```

static RandomHolmeKim(*n*, *m*, *p*, *seed=None*)

Return a random graph generated by the Holme and Kim algorithm for graphs with power law degree distribution and approximate average clustering.

INPUT:

- `n` – number of vertices
- `m` – number of random edges to add for each new node
- `p` – probability of adding a triangle after adding a random edge
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

From the NetworkX documentation: the average clustering has a hard time getting above a certain cutoff that depends on m. This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size. It is essentially the Barabasi-Albert growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle). This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired. It seems possible to have a disconnected graph with this algorithm since the initial m nodes may not be all linked to a new node on the first iteration like the BA model.

EXAMPLES:

We check that a random graph on 8 nodes with 2 random edges per node and a probability $p = 0.5$ of forming triangles contains a triangle:

```
sage: G = graphs.RandomHolmeKim(8, 2, 0.5)
sage: G.order(), G.size()
(8, 12)
sage: C3 = graphs.CycleGraph(3)
sage: G.subgraph_search(C3)
Subgraph of (): Graph on 3 vertices
```

```
sage: G = graphs.RandomHolmeKim(12, 3, .3)
sage: G.show()  # long time
```

REFERENCE:

[HK2002a]

static RandomIntervalGraph(*n*)

Returns a random interval graph.

An interval graph is built from a list $(a_i, b_i)_{1 \leq i \leq n}$ of intervals : to each interval of the list is associated one vertex, two vertices being adjacent if the two corresponding intervals intersect.

A random interval graph of order n is generated by picking random values for the (a_i, b_j), each of the two coordinates being generated from the uniform distribution on the interval $[0, 1]$.

This definitions follows [BF2001].

Note: The vertices are named 0, 1, 2, and so on. The intervals used to create the graph are saved with the graph and can be recovered using `get_vertex()` or `get_vertices()`.

INPUT:

- `n` (integer) – the number of vertices in the random graph.

EXAMPLES:

As for any interval graph, the chromatic number is equal to the clique number

```
sage: g = graphs.RandomIntervalGraph(8)
sage: g.clique_number() == g.chromatic_number()
True
```

static RandomLobster(*n*, *p*, *q*, *seed=None*)

Returns a random lobster.

A lobster is a tree that reduces to a caterpillar when pruning all leaf vertices. A caterpillar is a tree that reduces to a path when pruning all leaf vertices (q=0).

INPUT:

- `n` - expected number of vertices in the backbone
- `p` - probability of adding an edge to the backbone
- `q` - probability of adding an edge (claw) to the arms
- `seed` - a `random.Random` seed or a Python `int` for the random number generator (default: `None`).

EXAMPLES: We show the edge list of a random graph with 3 backbone nodes and probabilities $p = 0.7$ and $q = 0.3$:

```
sage: graphs.RandomLobster(3, 0.7, 0.3).edges(labels=False)
[]                                                                  # 32-bit
[(0, 1), (0, 5), (1, 2), (1, 6), (2, 3), (2, 7), (3, 4), (3, 8)]   # 64-bit
```

```
sage: G = graphs.RandomLobster(9, .6, .3)
sage: G.show()  # long time
```

static RandomNewmanWattsStrogatz(*n*, *k*, *p*, *seed=None*)

Return a Newman-Watts-Strogatz small world random graph on n vertices.

From the NetworkX documentation: first create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors. Then shortcuts are created by adding new edges as follows: for each edge $u - v$ in the underlying "n-ring with k nearest neighbors"; with probability p add a new edge $u - w$ with randomly-chosen existing node w. In contrast with `networkx.watts_strogatz_graph()`, no edges are removed.

INPUT:

- `n` – number of vertices
- `k` – each vertex is connected to its k nearest neighbors
- `p` – the probability of adding a new edge for each edge
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

EXAMPLES:

We check that the generated graph contains a cycle of order n:

```
sage: G = graphs.RandomNewmanWattsStrogatz(7, 2, 0.2)
sage: G.order(), G.size()   # py3
(7, 9)                      # 64-bit
(7, 10)                     # 32-bit
sage: G.order(), G.size()   # py2
(7, 9)
sage: C7 = graphs.CycleGraph(7)
sage: G.subgraph_search(C7)
Subgraph of (): Graph on 7 vertices
sage: G.diameter() <= C7.diameter()
True
```

```
sage: G = graphs.RandomNewmanWattsStrogatz(12, 2, .3)
sage: G.show()  # long time
```

REFERENCE:

[NWS2002]

static RandomRegular(*d*, *n*, *seed=None*)

Return a random d-regular graph on n vertices, or `False` on failure.

Since every edge is incident to two vertices, $n \times d$ must be even.

INPUT:

- `d` – degree
- `n` – number of vertices
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

EXAMPLES:

We check that a random graph with 8 nodes each of degree 3 is 3-regular:

```
sage: G = graphs.RandomRegular(3, 8)
sage: G.is_regular(k=3)
True
sage: G.degree_histogram()
[0, 0, 0, 8]
```

```
sage: G = graphs.RandomRegular(3, 20)
sage: if G:
....:     G.show()  # random output, long time
```

REFERENCES:

- [KV2003]
- [SW1999]

static RandomRegularBipartite(*n1*, *n2*, *d1*, *set_position=False*)

Return a random regular bipartite graph on $n1 + n2$ vertices.

The bipartite graph has $n1 * d1$ edges. Hence, $n2$ must divide $n1 * d1$. Each vertex of the set of cardinality $n1$ has degree $d1$ (which can be at most $n2$) and each vertex in the set of cardinality $n2$ has degree $(n1 * d1)/n2$. The bipartite graph has no multiple edges.

This generator implements an algorithm inspired by that of [MW1990] for the uniform generation of random regular bipartite graphs. It performs well when $d1 = o(n2^{1/3})$ or $(n2 - d1 = o(n2^{1/3}))$. In other cases, the running time can be huge. Note that the currently implemented algorithm does not generate uniformly random graphs.

INPUT:

- `n1, n2` – number of vertices in each side
- `d1` – degree of the vertices in the set of cardinality $n1$.
- `set_position` – boolean (default `False`); if set to `True`, we assign positions to the vertices so that the set of cardinality $n1$ is on the line $y = 1$ and the set of cardinality $n2$ is on the line $y = 0$.

EXAMPLES:

```
sage: g = graphs.RandomRegularBipartite(4, 6, 3)
sage: g.order(), g.size()
(10, 12)
sage: set(g.degree())
{2, 3}

sage: graphs.RandomRegularBipartite(1, 2, 2, set_position=True).get_pos()
{0: (1, 1.0), 1: (0, 0), 2: (2.0, 0.0)}
sage: graphs.RandomRegularBipartite(2, 1, 1, set_position=True).get_pos()
{0: (0, 1), 1: (2.0, 1.0), 2: (1, 0.0)}
sage: graphs.RandomRegularBipartite(2, 3, 3, set_position=True).get_pos()
{0: (0, 1), 1: (3.0, 1.0), 2: (0, 0), 3: (1.5, 0.0), 4: (3.0, 0.0)}
sage: graphs.RandomRegularBipartite(2, 3, 3, set_position=False).get_pos()
```

static RandomShell(*constructor*, *seed=None*)

Return a random shell graph for the constructor given.

INPUT:

- constructor – a list of 3-tuples (n, m, d), each representing a shell, where:
  - n – the number of vertices in the shell
  - m – the number of edges in the shell
  - d – the ratio of inter (next) shell edges to intra shell edges
- seed – a random.Random seed or a Python int for the random number generator (default: None)

EXAMPLES:

```
sage: G = graphs.RandomShell([(10,20,0.8),(20,40,0.8)])
sage: G.order(), G.size()
(30, 52)
sage: G.show()  # long time
```

static RandomToleranceGraph(n)

Returns a random tolerance graph.

The random tolerance graph is built from a random tolerance representation by using the function $ToleranceGraph$. This representation is a list $((l_0, r_0, t_0), (l_1, r_1, t_1), ..., (l_k, r_k, t_k))$ where $k = n - 1$ and $I_i = (l_i, r_i)$ denotes a random interval and t_i a random positive value. The width of the representation is limited to n**2 * 2**n.

Note: The vertices are named 0, 1, . . . , n-1. The tolerance representation used to create the graph is saved with the graph and can be recovered using get_vertex() or get_vertices().

INPUT:

- n – number of vertices of the random graph.

EXAMPLES:

Every tolerance graph is perfect. Hence, the chromatic number is equal to the clique number

```
sage: g = graphs.RandomToleranceGraph(8)
sage: g.clique_number() == g.chromatic_number()
True
```

static RandomTree(n)

Returns a random tree on n nodes numbered 0 through $n - 1$.

By Cayley's theorem, there are n^{n-2} trees with vertex set $\{0, 1, ..., n - 1\}$. This constructor chooses one of these uniformly at random.

ALGORITHM:

The algorithm works by generating an $(n - 2)$-long random sequence of numbers chosen independently and uniformly from $\{0, 1, \ldots, n - 1\}$ and then applies an inverse Prufer transformation.

INPUT:

- n - number of vertices in the tree

EXAMPLES:

```
sage: G = graphs.RandomTree(10)
sage: G.is_tree()
True
sage: G.show() # long time
```

static RandomTreePowerlaw(*n*, *gamma=3*, *tries=1000*, *seed=None*)
Return a tree with a power law degree distribution, or `False` on failure.

From the NetworkX documentation: a trial power law degree sequence is chosen and then elements are swapped with new elements from a power law distribution until the sequence makes a tree (size = order - 1).

INPUT:

- `n` – number of vertices
- `gamma` – exponent of power law distribution
- `tries` – number of attempts to adjust sequence to make a tree
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

EXAMPLES:

We check that the generated graph is a tree:

```
sage: G = graphs.RandomTreePowerlaw(10, 3)
sage: G.is_tree()
True
sage: G.order(), G.size()
(10, 9)
```

```
sage: G = graphs.RandomTreePowerlaw(15, 2)
sage: if G:
....:     G.show()  # random output, long time
```

static RandomTriangulation(*n*, *set_position=False*, *k=3*)
Return a random inner triangulation of an outer face of degree `k` with `n` vertices in total.

An inner triangulation is a plane graph all of whose faces (except the outer/unbounded face) are triangles (3-cycles).

INPUT:

- `n` – the number of vertices of the graph
- `k` – the size of the outer face
- `set_position` – boolean (default `False`); if set to `True`, this will compute coordinates for a planar drawing of the graph.

OUTPUT:

A random graph chosen uniformly among the inner triangulations of a *rooted* k-gon with n vertices (including the k vertices from the outer face). This is a planar graph and comes with a combinatorial embedding. The vertices of the root edge are labelled `-1` and `-2` and the outer face is the face returned by `Graph.faces()` in which `-1` and `-2` are consecutive vertices in this order.

Because some triangulations have nontrivial automorphism groups, this may not be equal to the uniform distribution among inner triangulations of unrooted k-gons.

ALGORITHM:

The algorithm is taken from [PS2006], Section 5.

Starting from a planar k-gonal forest (represented by its contour as a sequence of vertices), one performs local closures, until no one is possible. A local closure amounts to replace in the cyclic contour word a sequence `in1, in2, in3, lf, in3` by `in1, in3`.

At every step of the algorithm, newly created edges are recorded in a graph, which will be returned at the end. The combinatorial embedding is also computed and recorded in the output graph.

See also:

triangulations(), RandomTwoSphere().

EXAMPLES:

```
sage: G = graphs.RandomTriangulation(6, True); G
Graph on 6 vertices
sage: G.is_planar()
True
sage: G.girth()
3
sage: G.plot(vertex_size=0, vertex_labels=False)
Graphics object consisting of 13 graphics primitives

sage: H = graphs.RandomTriangulation(7, k=5)
sage: sorted(len(f) for f in H.faces())
[3, 3, 3, 3, 3, 3, 3, 5]
```

static RingedTree(*k*, *vertex_labels=True*)

Return the ringed tree on k-levels.

A ringed tree of level k is a binary tree with k levels (counting the root as a level), in which all vertices at the same level are connected by a ring.

More precisely, in each layer of the binary tree (i.e. a layer is the set of vertices $[2^i...2^{i+1}-1]$) two vertices u, v are adjacent if $u = v + 1$ or if $u = 2^i$ and $v =$.

Ringed trees are defined in [CFHM2013].

INPUT:

- k – the number of levels of the ringed tree.
- vertex_labels (boolean) – whether to label vertices as binary words (default) or as integers.

EXAMPLES:

```
sage: G = graphs.RingedTree(5)
sage: P = G.plot(vertex_labels=False, vertex_size=10)
sage: P.show() # long time
sage: G.vertices()
['', '0', '00', '000', '0000', '0001', '001', '0010', '0011', '01',
 '010', '0100', '0101', '011', '0110', '0111', '1', '10', '100',
 '1000', '1001', '101', '1010', '1011', '11', '110', '1100', '1101',
 '111', '1110', '1111']
```

static RobertsonGraph()

Return the Robertson graph.

See the Wikipedia article Robertson_graph.

EXAMPLES:

```
sage: g = graphs.RobertsonGraph()
sage: g.order()
19
sage: g.size()
38
```

(continues on next page)

(continued from previous page)

```
sage: g.diameter()
3
sage: g.girth()
5
sage: g.charpoly().factor()
(x - 4) * (x - 1)^2 * (x^2 + x - 5) * (x^2 + x - 1) * (x^2 - 3)^2 * (x^2 + x -
↪ 4)^2 * (x^2 + x - 3)^2
sage: g.chromatic_number()
3
sage: g.is_hamiltonian()
True
sage: g.is_vertex_transitive()
False
```

static RookGraph(*dim_list*, *radius=None*, *relabel=False*)

Returns the d-dimensional Rook's Graph with prescribed dimensions.

The 2-dimensional Rook's Graph of parameters n and m is a graph with nm vertices in which each vertex represents a square in an $n \times m$ chessboard, and each edge corresponds to a legal move by a rook.

The d-dimensional Rook Graph with $d >= 2$ has for vertex set the cells of a d-dimensional grid with prescribed dimensions, and each edge corresponds to a legal move by a rook in any of the dimensions.

The Rook's Graph for an $n \times m$ chessboard may also be defined as the Cartesian product of two complete graphs $K_n \Box K_m$.

INPUT:

- `dim_list` – an iterable object (list, set, dict) providing the dimensions $n_1, n_2, \ldots, n_d$, with $n_i \geq 1$, of the chessboard.
- `radius` – (default: `None`) by setting the radius to a positive integer, one may decrease the power of the rook to at most `radius` steps. When the radius is 1, the resulting graph is a d-dimensional grid.
- `relabel` – (default: `False`) a boolean set to `True` if vertices must be relabeled as integers.

EXAMPLES:

The (n, m)-Rook's Graph is isomorphic to the Cartesian product of two complete graphs:

```
sage: G = graphs.RookGraph( [3, 4] )
sage: H = ( graphs.CompleteGraph(3) ).cartesian_product( graphs.
↪CompleteGraph(4) )
sage: G.is_isomorphic( H )
True
```

When the radius is 1, the Rook's Graph is a grid:

```
sage: G = graphs.RookGraph( [3, 3, 4], radius=1 )
sage: H = graphs.GridGraph( [3, 4, 3] )
sage: G.is_isomorphic( H )
True
```

static SchlaefliGraph()

Return the Schläfli graph.

The Schläfli graph is the only strongly regular graphs of parameters $(27, 16, 10, 8)$ (see [GR2001]).

For more information, see the Wikipedia article Schläfli_graph.

See also:

Graph.is_strongly_regular() – tests whether a graph is strongly regular and/or returns its parameters.

Todo: Find a beautiful layout for this beautiful graph.

EXAMPLES:

Checking that the method actually returns the Schläfli graph:

```
sage: S = graphs.SchlaefliGraph()
sage: S.is_strongly_regular(parameters = True)
(27, 16, 10, 8)
```

The graph is vertex-transitive:

```
sage: S.is_vertex_transitive()
True
```

The neighborhood of each vertex is isomorphic to the complement of the Clebsch graph:

```
sage: neighborhood = S.subgraph(vertices = S.neighbors(0))
sage: graphs.ClebschGraph().complement().is_isomorphic(neighborhood)
True
```

static ShrikhandeGraph()

Return the Shrikhande graph.

For more information, see the MathWorld article on the Shrikhande graph or the Wikipedia article Shrikhande_graph.

See also:

Graph.is_strongly_regular() – tests whether a graph is strongly regular and/or returns its parameters.

EXAMPLES:

The Shrikhande graph was defined by S. S. Shrikhande in 1959. It has 16 vertices and 48 edges, and is strongly regular of degree 6 with parameters $(2, 2)$:

```
sage: G = graphs.ShrikhandeGraph(); G
Shrikhande graph: Graph on 16 vertices
sage: G.order()
16
sage: G.size()
48
sage: G.is_regular(6)
True
sage: set([ len([x for x in G.neighbors(i) if x in G.neighbors(j)])
....:       for i in range(G.order())
....:       for j in range(i) ])
{2}
```

It is non-planar, and both Hamiltonian and Eulerian:

```
sage: G.is_planar()
False
sage: G.is_hamiltonian()
```

(continues on next page)

(continued from previous page)

```
True
sage: G.is_eulerian()
True
```

It has radius 2, diameter 2, and girth 3:

```
sage: G.radius()
2
sage: G.diameter()
2
sage: G.girth()
3
```

Its chromatic number is 4 and its automorphism group is of order 192:

```
sage: G.chromatic_number()
4
sage: G.automorphism_group().cardinality()
192
```

It is an integral graph since it has only integral eigenvalues:

```
sage: G.characteristic_polynomial().factor()
(x - 6) * (x - 2)^6 * (x + 2)^9
```

It is a toroidal graph, and its embedding on a torus is dual to an embedding of the Dyck graph (*DyckGraph*).

static SierpinskiGasketGraph(n)

Return the Sierpinski Gasket graph of generation n.

All vertices but 3 have valence 4.

INPUT:

- n – an integer

OUTPUT:

a graph S_n with $3(3^{n-1}+1)/2$ vertices and 3^n edges, closely related to the famous Sierpinski triangle fractal.

All these graphs have a triangular shape, and three special vertices at top, bottom left and bottom right. These are the only vertices of valence 2, all the other ones having valence 4.

The graph S_1 (generation 1) is a triangle.

The graph S_{n+1} is obtained from the disjoint union of three copies A,B,C of S_n by identifying pairs of vertices: the top vertex of A with the bottom left vertex of B, the bottom right vertex of B with the top vertex of C, and the bottom left vertex of C with the bottom right vertex of A.

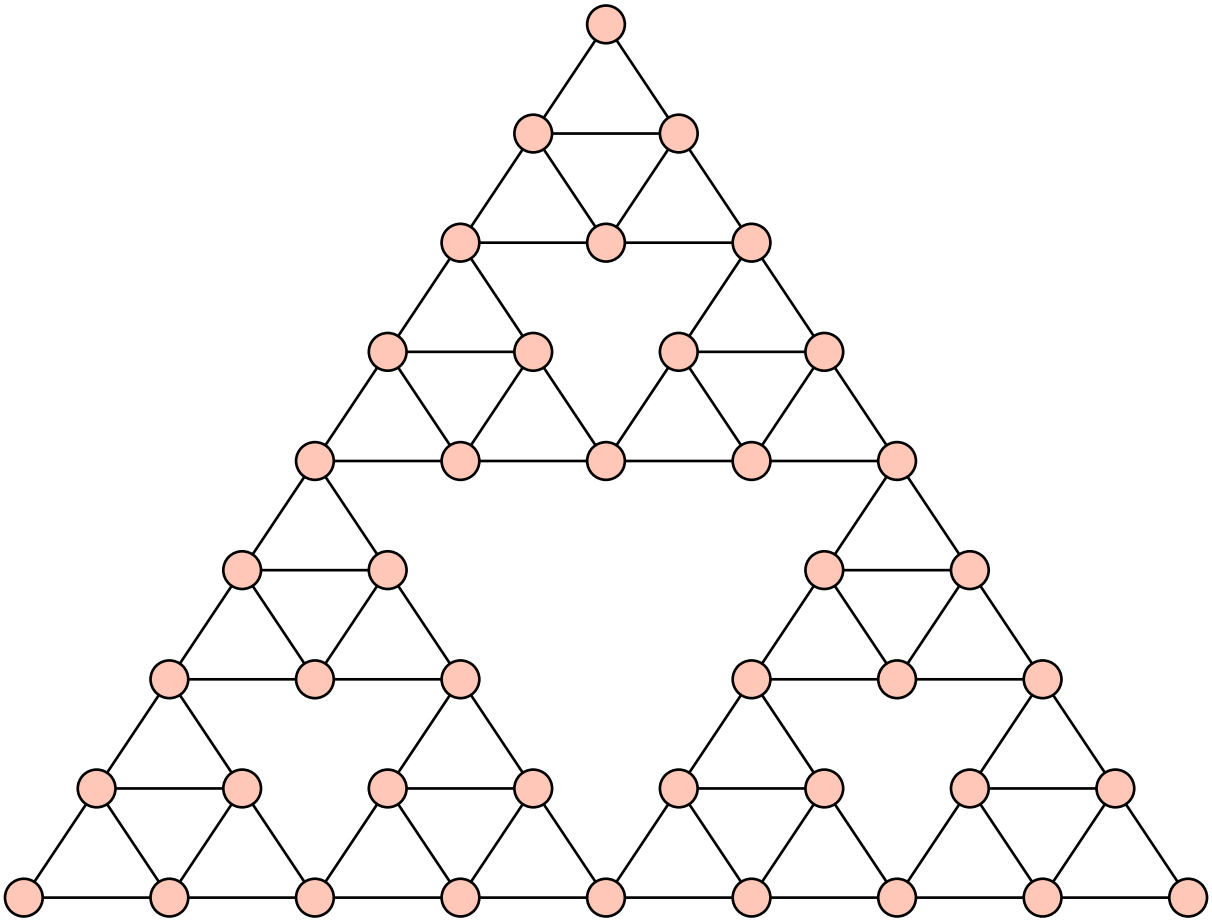
See also:

There is another familly of graphs called Sierpinski graphs, where all vertices but 3 have valence 3. They are available using `graphs.HanoiTowerGraph(3, n)`.

EXAMPLES:

```
sage: s4 = graphs.SierpinskiGasketGraph(4); s4
Graph on 42 vertices
```

(continues on next page)



(continued from previous page)

```
sage: s4.size()
81
sage: s4.degree_histogram()
[0, 0, 3, 0, 39]
sage: s4.is_hamiltonian()
True
```

REFERENCES:

[LLWC2011]

static SimsGewirtzGraph()

Return the Sims-Gewirtz Graph.

This graph is obtained from the Higman Sims graph by considering the graph induced by the vertices at distance two from the vertices of an (any) edge. It is the only strongly regular graph with parameters $v = 56$, $k = 10$, $\lambda = 0$, $\mu = 2$

For more information on the Sylvester graph, see https://www.win.tue.nl/~aeb/graphs/Sims-Gewirtz.html or its Wikipedia article Gewirtz_graph.

See also:

- *HigmanSimsGraph()*.

EXAMPLES:

```
sage: g = graphs.SimsGewirtzGraph(); g
Sims-Gewirtz Graph: Graph on 56 vertices
sage: g.order()
56
sage: g.size()
280
sage: g.is_strongly_regular(parameters = True)
(56, 10, 0, 2)
```

static SousselierGraph()

Return the Sousselier Graph.

The Sousselier graph is a hypohamiltonian graph on 16 vertices and 27 edges. For more information, see Wikipedia article Sousselier_graph or the corresponding French Wikipedia page.

EXAMPLES:

```
sage: g = graphs.SousselierGraph()
sage: g.order()
16
sage: g.size()
27
sage: g.radius()
2
sage: g.diameter()
3
sage: g.automorphism_group().cardinality()
2
sage: g.is_hamiltonian()
False
sage: g.delete_vertex(g.random_vertex())
```

(continues on next page)

(continued from previous page)

```
sage: g.is_hamiltonian()
True
```

static SquaredSkewHadamardMatrixGraph(n)

Pseudo-$OA(2n, 4n-1)$-graph from a skew Hadamard matrix of order $4n$

A strongly regular graph with parameters of the orthogonal array graph *OrthogonalArrayBlockGraph()*, also known as pseudo Latin squares graph $L_{2n}(4n-1)$, constructed from a skew Hadamard matrix of order $4n$, due to Goethals and Seidel, see [BL1984].

See also:

- *is_orthogonal_array_block_graph()*

EXAMPLES:

```
sage: graphs.SquaredSkewHadamardMatrixGraph(4).is_strongly_
↪regular(parameters=True)
(225, 112, 55, 56)
sage: graphs.SquaredSkewHadamardMatrixGraph(5).is_strongly_
↪regular(parameters=True)  # long time
(361, 180, 89, 90)
sage: graphs.SquaredSkewHadamardMatrixGraph(9).is_strongly_
↪regular(parameters=True)  # not tested
(1225, 612, 305, 306)
```

static StarGraph(n)

Return a star graph with $n+1$ nodes.

A Star graph is a basic structure where one node is connected to all other nodes.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each star graph will be displayed with the first (0) node in the center, the second node (1) at the top, with the rest following in a counterclockwise manner. (0) is the node connected to all other nodes.

The star graph is a good opportunity to compare efficiency of filling a position dictionary vs. using the spring-layout algorithm for plotting. As far as display, the spring-layout should push all other nodes away from the (0) node, and thus look very similar to this constructor's positioning.

EXAMPLES:

```
sage: import networkx
```

Compare the plots:

```
sage: n = networkx.star_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.StarGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

View many star graphs as a Sage Graphics Array

With this constructor (i.e., the position dictionary filled)

```
sage: g = []
sage: j = []
sage: for i in range(9):
```

(continues on next page)

(continued from previous page)

```
....:     k = graphs.StarGraph(i+3)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

Compared to plotting with the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.star_graph(i+3)
....:     k = Graph(spr)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static SuzukiGraph()

Return the Suzuki Graph.

The Suzuki graph has 1782 vertices, and is strongly regular with parameters $(1782, 416, 100, 96)$. Known as S.15 in [Hub1975].

Note: It takes approximately 50 seconds to build this graph. Do not be too impatient.

EXAMPLES:

```
sage: g = graphs.SuzukiGraph(); g             # optional internet # not tested
Suzuki graph: Graph on 1782 vertices
sage: g.is_strongly_regular(parameters=True) # optional internet # not tested
(1782, 416, 100, 96)
```

static SwitchedSquaredSkewHadamardMatrixGraph(n)

A strongly regular graph in Seidel switching class of $SquaredSkewHadamardMatrixGraph$

A strongly regular graph in the *Seidel switching* class of the disjoint union of a 1-vertex graph and the one produced by *Pseudo-L_{2n}(4n-1)*

In this case, the other possible parameter set of a strongly regular graph in the Seidel switching class of the latter graph (see [**?**]) coincides with the set of parameters of the complement of the graph returned by this function.

See also:

- *is_switch_skewhad()*

EXAMPLES:

```
sage: g=graphs.SwitchedSquaredSkewHadamardMatrixGraph(4)
sage: g.is_strongly_regular(parameters=True)
(226, 105, 48, 49)
sage: from sage.combinat.designs.twographs import twograph_descendant
sage: twograph_descendant(g,0).is_strongly_regular(parameters=True)
(225, 112, 55, 56)
sage: twograph_descendant(g.complement(),0).is_strongly_
↪regular(parameters=True)
(225, 112, 55, 56)
```

static SylvesterGraph()

Return the Sylvester Graph.

This graph is obtained from the Hoffman Singleton graph by considering the graph induced by the vertices at distance two from the vertices of an (any) edge.

For more information on the Sylvester graph, see https://www.win.tue.nl/~aeb/graphs/Sylvester.html.

See also:

- *HoffmanSingletonGraph()*.

EXAMPLES:

```
sage: g = graphs.SylvesterGraph(); g
Sylvester Graph: Graph on 36 vertices
sage: g.order()
36
sage: g.size()
90
sage: g.is_regular(k=5)
True
```

static SymplecticDualPolarGraph(m, q)

Returns the Symplectic Dual Polar Graph $DSp(m, q)$.

For more information on Symplectic Dual Polar graphs, see [BCN1989] and Sect. 2.3.1 of [Coh1981].

INPUT:

- m, q (integers) – q must be a prime power, and m must be even.

EXAMPLES:

```
sage: G = graphs.SymplecticDualPolarGraph(6,3); G       # not tested (long␣
↪time)
Symplectic Dual Polar Graph DSp(6, 3): Graph on 1120 vertices
sage: G.is_distance_regular(parameters=True)            # not tested (long␣
↪time)
([39, 36, 27, None], [None, 1, 4, 13])
```

static SymplecticPolarGraph(d, q, *algorithm=None*)

Returns the Symplectic Polar Graph $Sp(d, q)$.

The Symplectic Polar Graph $Sp(d, q)$ is built from a projective space of dimension $d - 1$ over a field F_q, and a symplectic form f. Two vertices u, v are made adjacent if $f(u, v) = 0$.

See the page on symplectic graphs on Andries Brouwer's website.

INPUT:

- d, q (integers) – note that only even values of d are accepted by the function.
- algorithm – if set to 'gap' then the computation is carried via GAP library interface, computing totally singular subspaces, which is faster for $q > 3$. Otherwise it is done directly.

EXAMPLES:

Computation of the spectrum of $Sp(6, 2)$:

```
sage: g = graphs.SymplecticPolarGraph(6,2)
sage: g.is_strongly_regular(parameters=True)
(63, 30, 13, 15)
sage: set(g.spectrum()) == {-5, 3, 30}
True
```

The parameters of $Sp(4, q)$ are the same as of $O(5, q)$, but they are not isomorphic if q is odd:

```
sage: G = graphs.SymplecticPolarGraph(4,3)
sage: G.is_strongly_regular(parameters=True)
(40, 12, 2, 4)
sage: O=graphs.OrthogonalPolarGraph(5,3)
sage: O.is_strongly_regular(parameters=True)
(40, 12, 2, 4)
sage: O.is_isomorphic(G)
False
sage: graphs.SymplecticPolarGraph(6,4,algorithm="gap").is_strongly_
↪regular(parameters=True) # not tested (long time)
(1365, 340, 83, 85)
```

static SzekeresSnarkGraph()

Return the Szekeres Snark Graph.

The Szekeres graph is a snark with 50 vertices and 75 edges. For more information on this graph, see the Wikipedia article Szekeres_snark.

EXAMPLES:

```
sage: g = graphs.SzekeresSnarkGraph()
sage: g.order()
50
sage: g.size()
75
sage: g.chromatic_number()
3
```

static T2starGeneralizedQuadrangleGraph(*q*, *dual=False*, *hyperoval=None*, *field=None*, *check_hyperoval=True*)

Return the collinearity graph of the generalized quadrangle $T_2^*(q)$, or of its dual

Let $q = 2^k$ and $\Theta = PG(3, q)$. $T_2^*(q)$ is a generalized quadrangle (Wikipedia article Generalized_quadrangle) of order $(q - 1, q + 1)$, see 3.1.3 in [PT2009]. Fix a plane $\Pi \subset \Theta$ and a hyperoval $O \subset \Pi$. The points of $T_2^*(q) := T_2^*(O)$ are the points of Θ outside Π, and the lines are the lines of Θ outside Π that meet Π in a point of O.

INPUT:

- q – a power of two
- dual – if False (default), return the graph of $T_2^*(O)$. Otherwise return the graph of the dual $T_2^*(O)$.
- hyperoval – a hyperoval (i.e. a complete 2-arc; a set of points in the plane meeting every line in 0 or 2 points) in the plane of points with 0th coordinate 0 in $PG(3, q)$ over the field field. Each point of

`hyperoval` must be a length 4 vector over `field` with 1st non-0 coordinate equal to 1. By default, `hyperoval` and `field` are not specified, and constructed on the fly. In particular, `hyperoval` we build is the classical one, i.e. a conic with the point of intersection of its tangent lines.

- `field` – an instance of a finite field of order q, must be provided if `hyperoval` is provided.
- `check_hyperoval` – (default: `True`) if `True`, check `hyperoval` for correctness.

EXAMPLES:

using the built-in construction:

```
sage: g=graphs.T2starGeneralizedQuadrangleGraph(4); g
T2*(O,4); GQ(3, 5): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 18, 2, 6)
sage: g=graphs.T2starGeneralizedQuadrangleGraph(4,dual=True); g
T2*(O,4)*; GQ(5, 3): Graph on 96 vertices
sage: g.is_strongly_regular(parameters=True)
(96, 20, 4, 4)
```

supplying your own hyperoval:

```
sage: F=GF(4,'b')
sage: O=[vector(F,(0,0,0,1)),vector(F,(0,0,1,0))]+[vector(F, (0,1,x^2,x)) for␣
↪x in F]
sage: g=graphs.T2starGeneralizedQuadrangleGraph(4, hyperoval=O, field=F); g
T2*(O,4); GQ(3, 5): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 18, 2, 6)
```

static TadpoleGraph(*n1*, *n2*)

Return a tadpole graph with n1+n2 nodes.

A tadpole graph is a path graph (order n2) connected to a cycle graph (order n1).

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the cycle graph will be drawn in the lower-left corner with the (n1)th node at a 45 degree angle above the right horizontal center of the cycle graph, leading directly into the path graph.

EXAMPLES:

Construct and show a tadpole graph Cycle = 13, Stick = 4:

```
sage: g = graphs.TadpoleGraph(13, 4); g
Tadpole graph: Graph on 17 vertices
sage: g.show() # long time
```

static TaylorTwographDescendantSRG(*q*, *clique_partition=None*)

constructing the descendant graph of the Taylor's two-graph for $U_3(q)$, q odd

This is a strongly regular graph with parameters $(v,k,\lambda,\mu) = (q^3,(q^2+1)(q-1)/2,(q-1)^3/4-1,(q^2+1)(q-1)/4)$ obtained as a two-graph descendant of the `Taylor's two-graph` T. This graph admits a partition into cliques of size q, which are useful in *TaylorTwographSRG()*, a strongly regular graph on q^3+1 vertices in the Seidel switching class of T, for which we need $(q^2+1)/2$ cliques. The cliques are the q^2 lines on v_0 of the projective plane containing the unital for $U_3(q)$, and intersecting the unital (i.e. the vertices of the graph and the point we remove) in $q+1$ points. This is all taken from §7E of [BL1984].

INPUT:

- `q` – a power of an odd prime number

- clique_partition – if True, return $q^2 - 1$ cliques of size q with empty pairwise intersection. (Removing all of them leaves a clique, too), and the point removed from the unital.

EXAMPLES:

```
sage: g=graphs.TaylorTwographDescendantSRG(3); g
Taylor two-graph descendant SRG: Graph on 27 vertices
sage: g.is_strongly_regular(parameters=True)
(27, 10, 1, 5)
sage: from sage.combinat.designs.twographs import taylor_twograph
sage: T = taylor_twograph(3)                                # long time
sage: g.is_isomorphic(T.descendant(T.ground_set()[1]))  # long time
True
sage: g=graphs.TaylorTwographDescendantSRG(5)     # not tested (long time)
sage: g.is_strongly_regular(parameters=True)   # not tested (long time)
(125, 52, 15, 26)
```

static TaylorTwographSRG(q)

constructing a strongly regular graph from the Taylor's two-graph for $U_3(q)$, q odd

This is a strongly regular graph with parameters $(v, k, \lambda, \mu) = (q^3+1, q(q^2+1)/2, (q^2+3)(q-1)/4, (q^2+1)(q+1)/4)$ in the Seidel switching class of Taylor two-graph. Details are in §7E of [BL1984].

INPUT:

- q – a power of an odd prime number

See also:

- *TaylorTwographDescendantSRG()*

EXAMPLES:

```
sage: t=graphs.TaylorTwographSRG(3); t
Taylor two-graph SRG: Graph on 28 vertices
sage: t.is_strongly_regular(parameters=True)
(28, 15, 6, 10)
```

static TetrahedralGraph()

Returns a tetrahedral graph (with 4 nodes).

A tetrahedron is a 4-sided triangular pyramid. The tetrahedral graph corresponds to the connectivity of the vertices of the tetrahedron. This graph is equivalent to a wheel graph with 4 nodes and also a complete graph on four nodes. (See examples below).

PLOTTING: The tetrahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Tetrahedral graph

```
sage: g = graphs.TetrahedralGraph()
sage: g.show() # long time
```

The following example requires networkx:

```
sage: import networkx as NX
```

Compare this Tetrahedral, Wheel(4), Complete(4), and the Tetrahedral plotted with the spring-layout algorithm below in a Sage graphics array:

```
sage: tetra_pos = graphs.TetrahedralGraph()
sage: tetra_spring = Graph(NX.tetrahedral_graph())
sage: wheel = graphs.WheelGraph(4)
sage: complete = graphs.CompleteGraph(4)
sage: g = [tetra_pos, tetra_spring, wheel, complete]
sage: j = []
sage: for i in range(2):
....:     n = []
....:     for m in range(2):
....:         n.append(g[i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show() # long time
```

static ThomsenGraph()

Return the Thomsen Graph.

The Thomsen Graph is actually a complete bipartite graph with $(n1, n2) = (3, 3)$. It is also called the Utility graph.

PLOTTING: See CompleteBipartiteGraph.

EXAMPLES:

```
sage: T = graphs.ThomsenGraph()
sage: T
Thomsen graph: Graph on 6 vertices
sage: T.graph6_string()
'EFz_'
sage: (graphs.ThomsenGraph()).show() # long time
```

static TietzeGraph()

Return the Tietze Graph.

For more information on the Tietze Graph, see the Wikipedia article Tietze's_graph.

EXAMPLES:

```
sage: g = graphs.TietzeGraph()
sage: g.order()
12
sage: g.size()
18
sage: g.diameter()
3
sage: g.girth()
3
sage: g.automorphism_group().cardinality()
12
sage: g.automorphism_group().is_isomorphic(groups.permutation.Dihedral(6))
True
```

static ToleranceGraph(*tolrep*)

Return the graph generated by the tolerance representation `tolrep`.

The tolerance representation `tolrep` is described by the list $((l_0, r_0, t_0), (l_1, r_1, t_1), ..., (l_k, r_k, t_k))$ where $I_i = (l_i, r_i)$ denotes a closed interval on the real line with $l_i < r_i$ and t_i a positive value, called toler-

ance. This representation generates the tolerance graph with the vertex set {0,1, ..., k} and the edge set $(i, j) : |I_i \cap I_j| \geq \min t_i, t_j$ where $|I_i \cap I_j|$ denotes the length of the intersection of I_i and I_j.

INPUT:

- `tolrep` – list of triples (l_i, r_i, t_i) where (l_i, r_i) denotes a closed interval on the real line and t_i a positive value.

Note: The vertices are named 0, 1, ..., k. The tolerance representation used to create the graph is saved with the graph and can be recovered using `get_vertex()` or `get_vertices()`.

EXAMPLES:

The following code creates a tolerance representation `tolrep`, generates its tolerance graph `g`, and applies some checks:

```
sage: tolrep = [(1,4,3),(1,2,1),(2,3,1),(0,3,3)]
sage: g = graphs.ToleranceGraph(tolrep)
sage: g.get_vertex(3)
(0, 3, 3)
sage: neigh = g.neighbors(3)
sage: for v in neigh: print(g.get_vertex(v))
(1, 2, 1)
(2, 3, 1)
sage: g.is_interval()
False
sage: g.is_weakly_chordal()
True
```

The intervals in the list need not be distinct

```
sage: tolrep2 = [(0,4,5),(1,2,1),(2,3,1),(0,4,5)]
sage: g2 = graphs.ToleranceGraph(tolrep2)
sage: g2.get_vertices()
{0: (0, 4, 5), 1: (1, 2, 1), 2: (2, 3, 1), 3: (0, 4, 5)}
sage: g2.is_isomorphic(g)
True
```

Real values are also allowed

```
sage: tolrep = [(0.1,3.3,4.4),(1.1,2.5,1.1),(1.4,4.4,3.3)]
sage: g = graphs.ToleranceGraph(tolrep)
sage: g.is_isomorphic(graphs.PathGraph(3))
True
```

static Toroidal6RegularGrid2dGraph(*n1*, *n2*)

Returns a toroidal 6-regular grid.

The toroidal 6-regular grid is a 6-regular graph on $n_1 \times n_2$ vertices and its elements have coordinates (i, j) for $i \in \{0...i-1\}$ and $j \in \{0...j-1\}$.

Its edges are those of the *ToroidalGrid2dGraph()*, to which are added the edges between (i, j) and $((i+1)\%n_1, (j+1)\%n_2)$.

INPUT:

- `n1, n2` (integers) – see above.

EXAMPLES:

The toroidal 6-regular grid on 25 elements:

```
sage: g = graphs.Toroidal6RegularGrid2dGraph(5,5)
sage: g.is_regular(k=6)
True
sage: g.is_vertex_transitive()
True
sage: g.line_graph().is_vertex_transitive()
True
sage: g.automorphism_group().cardinality()
300
sage: g.is_hamiltonian()
True
```

static ToroidalGrid2dGraph(*n1*, *n2*)

Returns a toroidal 2-dimensional grid graph with $n_1 n_2$ nodes (n_1 rows and n_2 columns).

The toroidal 2-dimensional grid with parameters n_1, n_2 is the 2-dimensional grid graph with identical parameters to which are added the edges $((i, 0), (i, n_2 - 1))$ and $((0, i), (n_1 - 1, i))$.

EXAMPLES:

The toroidal 2-dimensional grid is a regular graph, while the usual 2-dimensional grid is not

```
sage: tgrid = graphs.ToroidalGrid2dGraph(8,9)
sage: print(tgrid)
Toroidal 2D Grid Graph with parameters 8,9
sage: grid = graphs.Grid2dGraph(8,9)
sage: grid.is_regular()
False
sage: tgrid.is_regular()
True
```

static TruncatedIcosidodecahedralGraph()

Return the truncated icosidodecahedron.

The truncated icosidodecahedron is an Archimedean solid with 30 square faces, 20 regular hexagonal faces, 12 regular decagonal faces, 120 vertices and 180 edges. For more information, see the Wikipedia article Truncated_icosidodecahedron.

EXAMPLES:

Unfortunately, this graph can not be constructed currently, due to numerical issues:

```
sage: g = graphs.TruncatedIcosidodecahedralGraph(); g
Traceback (most recent call last):
...
ValueError: *Error: Numerical inconsistency is found.  Use the GMP exact␣
↪arithmetic.
sage: g.order(), g.size() # not tested
(120, 180)
```

static TruncatedTetrahedralGraph()

Return the truncated tetrahedron.

The truncated tetrahedron is an Archimedean solid with 12 vertices and 18 edges. For more information, see the Wikipedia article Truncated_tetrahedron.

EXAMPLES:

```
sage: g = graphs.TruncatedTetrahedralGraph(); g
Truncated Tetrahedron: Graph on 12 vertices
sage: g.order(), g.size()
(12, 18)
sage: g.is_isomorphic(polytopes.simplex(3).truncation().graph())
True
```

static TuranGraph(*n*, *r*)

Returns the Turan graph with parameters n, r.

Turan graphs are complete multipartite graphs with n vertices and r subsets, denoted $T(n, r)$, with the property that the sizes of the subsets are as close to equal as possible. The graph $T(n, r)$ will have $n \pmod r$ subsets of size $\lfloor n/r \rfloor$ and $r - (n \pmod r)$ subsets of size $\lceil n/r \rceil$. See the Wikipedia article Turan_graph for more information.

INPUT:

- `n` (integer)– the number of vertices in the graph.
- `r` (integer) – the number of partitions of the graph.

EXAMPLES:

The Turan graph is a complete multipartite graph.

```
sage: g = graphs.TuranGraph(13, 4)
sage: k = graphs.CompleteMultipartiteGraph([3,3,3,4])
sage: g.is_isomorphic(k)
True
```

The Turan graph $T(n, r)$ has $\lfloor \frac{(r-1)(n^2)}{2r} \rfloor$ edges.

```
sage: n = 13
sage: r = 4
sage: g = graphs.TuranGraph(n,r)
sage: g.size() == floor((r-1)*(n**2)/(2*r))
True
```

static Tutte12Cage()

Return the Tutte 12-Cage.

See the Wikipedia article Tutte_12-cage.

EXAMPLES:

```
sage: g = graphs.Tutte12Cage()
sage: g.order()
126
sage: g.size()
189
sage: g.girth()
12
sage: g.diameter()
6
sage: g.show()
```

static TutteCoxeterGraph(*embedding=2*)

Return the Tutte-Coxeter graph.

See the Wikipedia article Tutte-Coxeter_graph.

INPUT:

- `embedding` – two embeddings are available, and can be selected by setting `embedding` to 1 or 2.

EXAMPLES:

```
sage: g = graphs.TutteCoxeterGraph()
sage: g.order()
30
sage: g.size()
45
sage: g.girth()
8
sage: g.diameter()
4
sage: g.show()
sage: graphs.TutteCoxeterGraph(embedding=1).show()
```

static TutteGraph()

Return the Tutte Graph.

The Tutte graph is a 3-regular, 3-connected, and planar non-hamiltonian graph. For more information on the Tutte Graph, see the Wikipedia article Tutte_graph.

EXAMPLES:

```
sage: g = graphs.TutteGraph()
sage: g.order()
46
sage: g.size()
69
sage: g.is_planar()
True
sage: g.vertex_connectivity() # long time
3
sage: g.girth()
4
sage: g.automorphism_group().cardinality()
3
sage: g.is_hamiltonian()
False
```

static U42Graph216()

Return a (216,40,4,8)-strongly regular graph from [CRS2016].

Build the graph, interpreting the $U_4(2)$-action considered in [CRS2016] as the one on the hyperbolic lines of the corresponding unitary polar space, and then doing the unique merging of the orbitals leading to a graph with the parameters in question.

EXAMPLES:

```
sage: G=graphs.U42Graph216()                  # optional - gap_packages (grape)
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages (grape)
(216, 40, 4, 8)
```

static U42Graph540()

Return a (540,187,58,68)-strongly regular graph from [CRS2016].

Build the graph, interpreting the $U_4(2)$-action considered in [CRS2016] as the action of $U_4(2) = Sp_4(3) < U_4(3)$ on the nonsingular, w.r.t. to the Hermitean form stabilised by $U_4(3)$, points of the 3-dimensional pro-

jective space over $GF(9)$. There are several possible mergings of orbitals, some leading to non-isomorphic graphs with the same parameters. We found the merging here using [FK1991].

EXAMPLES:

```
sage: G=graphs.U42Graph540() # optional - gap_packages (grape)
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages (grape)
(540, 187, 58, 68)
```

static USAMap(*continental=False*)

Return states of USA as a graph of common border.

The graph has an edge between those states that have common *land* border line or point. Hence for example Colorado and Arizona are marked as neighbors, but Michigan and Minnesota are not.

INPUT:

- `continental`, a Boolean – if set, exclude Alaska and Hawaii

EXAMPLES:

How many states are neighbor's neighbor for Pennsylvania:

```
sage: USA = graphs.USAMap()
sage: len([n2 for n2 in USA if USA.distance('Pennsylvania', n2) == 2])
7
```

Diameter for continental USA:

```
sage: USAcont = graphs.USAMap(continental=True)
sage: USAcont.diameter()
11
```

static UnitaryDualPolarGraph(m, q)

Returns the Dual Unitary Polar Graph $U(m, q)$.

For more information on Unitary Dual Polar graphs, see [BCN1989] and Sect. 2.3.1 of [Coh1981].

INPUT:

- `m`, `q` (integers) – q must be a prime power.

EXAMPLES:

The point graph of a generalized quadrangle (see Wikipedia article Generalized_quadrangle, [PT2009]) of order (8,4):

```
sage: G = graphs.UnitaryDualPolarGraph(5,2); G    # long time
Unitary Dual Polar Graph DU(5, 2); GQ(8, 4): Graph on 297 vertices
sage: G.is_strongly_regular(parameters=True)      # long time
(297, 40, 7, 5)
```

Another way to get the generalized quadrangle of order (2,4):

```
sage: G = graphs.UnitaryDualPolarGraph(4,2); G
Unitary Dual Polar Graph DU(4, 2); GQ(2, 4): Graph on 27 vertices
sage: G.is_isomorphic(graphs.OrthogonalPolarGraph(6,2,'-'))
True
```

A bigger graph:

```
sage: G = graphs.UnitaryDualPolarGraph(6,2); G   # not tested (long time)
Unitary Dual Polar Graph DU(6, 2): Graph on 891 vertices
sage: G.is_distance_regular(parameters=True)     # not tested (long time)
([42, 40, 32, None], [None, 1, 5, 21])
```

static UnitaryPolarGraph(*m*, *q*, *algorithm='gap'*)

Returns the Unitary Polar Graph $U(m, q)$.

For more information on Unitary Polar graphs, see the page of Andries Brouwer's website.

INPUT:

- `m, q` (integers) – q must be a prime power.
- `algorithm` – if set to 'gap' then the computation is carried via GAP library interface, computing totally singular subspaces, which is faster for large examples (especially with $q > 2$). Otherwise it is done directly.

EXAMPLES:

```
sage: G = graphs.UnitaryPolarGraph(4,2); G
Unitary Polar Graph U(4, 2); GQ(4, 2): Graph on 45 vertices
sage: G.is_strongly_regular(parameters=True)
(45, 12, 3, 3)
sage: graphs.UnitaryPolarGraph(5,2).is_strongly_regular(parameters=True)
(165, 36, 3, 9)
sage: graphs.UnitaryPolarGraph(6,2)     # not tested (long time)
Unitary Polar Graph U(6, 2): Graph on 693 vertices
```

static WagnerGraph()

Return the Wagner Graph.

See the Wikipedia article Wagner_graph.

EXAMPLES:

```
sage: g = graphs.WagnerGraph()
sage: g.order()
8
sage: g.size()
12
sage: g.girth()
4
sage: g.diameter()
2
sage: g.show()
```

static WatkinsSnarkGraph()

Return the Watkins Snark Graph.

The Watkins Graph is a snark with 50 vertices and 75 edges. For more information, see the Wikipedia article Watkins_snark.

EXAMPLES:

```
sage: g = graphs.WatkinsSnarkGraph()
sage: g.order()
50
sage: g.size()
75
```

(continues on next page)

(continued from previous page)

```
sage: g.chromatic_number()
3
```

static WellsGraph()

Return the Wells graph.

For more information on the Wells graph (also called Armanios-Wells graph), see this page.

The implementation follows the construction given on page 266 of [BCN1989]. This requires to create intermediate graphs and run a small isomorphism test, while everything could be replaced by a pre-computed list of edges : I believe that it is better to keep "the recipe" in the code, however, as it is quite unlikely that this could become the most time-consuming operation in any sensible algorithm, and .... "preserves knowledge", which is what open-source software is meant to do.

EXAMPLES:

```
sage: g = graphs.WellsGraph(); g
Wells graph: Graph on 32 vertices
sage: g.order()
32
sage: g.size()
80
sage: g.girth()
5
sage: g.diameter()
4
sage: g.chromatic_number()
4
sage: g.is_regular(k=5)
True
```

static WheelGraph(*n*)

Returns a Wheel graph with n nodes.

A Wheel graph is a basic structure where one node is connected to all other nodes and those (outer) nodes are connected cyclically.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each wheel graph will be displayed with the first (0) node in the center, the second node at the top, and the rest following in a counterclockwise manner.

With the wheel graph, we see that it doesn't take a very large n at all for the spring-layout to give a counter-intuitive display. (See Graphics Array examples below).

EXAMPLES:

We view many wheel graphs with a Sage Graphics Array, first with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:  k = graphs.WheelGraph(i+3)
....:  g.append(k)
...
sage: for i in range(3):
....:  n = []
....:  for m in range(3):
```

(continues on next page)

(continued from previous page)

```
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
...
sage: G = graphics_array(j)
sage: G.show() # long time
```

Next, using the spring-layout algorithm:

```
sage: import networkx
sage: g = []
sage: j = []
sage: for i in range(9):
....:     spr = networkx.wheel_graph(i+3)
....:     k = Graph(spr)
....:     g.append(k)
...
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
...
sage: G = graphics_array(j)
sage: G.show() # long time
```

Compare the plotting:

```
sage: n = networkx.wheel_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.WheelGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

static WienerArayaGraph()

Return the Wiener-Araya Graph.

The Wiener-Araya Graph is a planar hypohamiltonian graph on 42 vertices and 67 edges. For more information, see the Wolfram Page on the Wiener-Araya Graph or Wikipedia article Wiener-Araya_graph.

EXAMPLES:

```
sage: g = graphs.WienerArayaGraph()
sage: g.order()
42
sage: g.size()
67
sage: g.girth()
4
sage: g.is_planar()
True
sage: g.is_hamiltonian() # not tested -- around 30s long
False
sage: g.delete_vertex(g.random_vertex())
sage: g.is_hamiltonian()
True
```

static WindmillGraph(k, n)

Return the Windmill graph $Wd(k, n)$.

The windmill graph $Wd(k, n)$ is an undirected graph constructed for $k \geq 2$ and $n \geq 2$ by joining n copies of the complete graph K_k at a shared vertex. It has $(k-1)n+1$ vertices and $nk(k-1)/2$ edges, girth 3 (if $k > 2$), radius 1 and diameter 2. It has vertex connectivity 1 because its central vertex is an articulation point; however, like the complete graphs from which it is formed, it is $(k-1)$-edge-connected. It is trivially perfect and a block graph.

See also:

- Wikipedia article Windmill_graph
- *GraphGenerators.StarGraph()*
- *GraphGenerators.FriendshipGraph()*

EXAMPLES:

The Windmill graph $Wd(2, n)$ is a star graph:

```
sage: n = 5
sage: W = graphs.WindmillGraph(2, n)
sage: W.is_isomorphic( graphs.StarGraph(n) )
True
```

The Windmill graph $Wd(3, n)$ is the Friendship graph F_n:

```
sage: n = 5
sage: W = graphs.WindmillGraph(3, n)
sage: W.is_isomorphic( graphs.FriendshipGraph(n) )
True
```

The Windmill graph $Wd(3, 2)$ is the Butterfly graph:

```
sage: W = graphs.WindmillGraph(3, 2)
sage: W.is_isomorphic( graphs.ButterflyGraph() )
True
```

The Windmill graph $Wd(k, n)$ has chromatic number k:

```
sage: n,k = 5,6
sage: W = graphs.WindmillGraph(k, n)
sage: W.chromatic_number() == k
True
```

static WorldMap()

Returns the Graph of all the countries, in which two countries are adjacent in the graph if they have a common boundary.

This graph has been built from the data available in The CIA World Factbook [CIA] (2009-08-21).

The returned graph `G` has a member `G.gps_coordinates` equal to a dictionary containing the GPS coordinates of each country's capital city.

EXAMPLES:

```
sage: g = graphs.WorldMap()
sage: g.has_edge("France", "Italy")
True
sage: g.gps_coordinates["Bolivia"]
[[17, 'S'], [65, 'W']]
```

(continues on next page)

(continued from previous page)

```
sage: sorted(g.connected_component_containing_vertex('Ireland'))
['Ireland', 'United Kingdom']
```

REFERENCE:

[CIA]

static chang_graphs()

Return the three Chang graphs.

Three of the four strongly regular graphs of parameters $(28, 12, 6, 4)$ are called the Chang graphs. The fourth is the line graph of K_8. For more information about the Chang graphs, see the Wikipedia article Chang_graphs or https://www.win.tue.nl/~aeb/graphs/Chang.html.

EXAMPLES: check that we get 4 non-isomorphic s.r.g.'s with the same parameters:

```
sage: chang_graphs = graphs.chang_graphs()
sage: K8 = graphs.CompleteGraph(8)
sage: T8 = K8.line_graph()
sage: four_srg = chang_graphs + [T8]
sage: for g in four_srg:
....:     print(g.is_strongly_regular(parameters=True))
(28, 12, 6, 4)
(28, 12, 6, 4)
(28, 12, 6, 4)
(28, 12, 6, 4)
sage: from itertools import combinations
sage: for g1,g2 in combinations(four_srg,2):
....:     assert not g1.is_isomorphic(g2)
```

Construct the Chang graphs by Seidel switching:

```
sage: c3c5=graphs.CycleGraph(3).disjoint_union(graphs.CycleGraph(5))
sage: c8=graphs.CycleGraph(8)
sage: s=[K8.subgraph_search(c8).edges(),
....:    [(0,1,None),(2,3,None),(4,5,None),(6,7,None)],
....:    K8.subgraph_search(c3c5).edges()]
sage: list(map(lambda x,G: T8.seidel_switching(x, inplace=False).is_
↪isomorphic(G),
....:                  s, chang_graphs))
[True, True, True]
```

cospectral_graphs(*vertices*, *matrix_function=<function GraphGenerators.<lambda> at 0x7f7e490fe400>*, *graphs=None*)

Find all sets of graphs on `vertices` vertices (with possible restrictions) which are cospectral with respect to a constructed matrix.

INPUT:

- `vertices` - The number of vertices in the graphs to be tested
- `matrix_function` - A function taking a graph and giving back a matrix. This defaults to the adjacency matrix. The spectra examined are the spectra of these matrices.
- `graphs` - One of three things:
  - `None` (default) - test all graphs having `vertices` vertices
  - a function taking a graph and returning `True` or `False` - test only the graphs on `vertices` vertices for which the function returns `True`

– a list of graphs (or other iterable object) - these graphs are tested for cospectral sets. In this case, `vertices` is ignored.

OUTPUT:

A list of lists of graphs. Each sublist will be a list of cospectral graphs (lists of cardinality 1 being omitted).

See also:

Graph.is_strongly_regular() – tests whether a graph is strongly regular and/or returns its parameters.

EXAMPLES:

```
sage: g=graphs.cospectral_graphs(5)
sage: sorted(sorted(g.graph6_string() for g in glist) for glist in g)
[['Dr?', 'Ds_']]
sage: g[0][1].am().charpoly()==g[0][1].am().charpoly()
True
```

There are two sets of cospectral graphs on six vertices with no isolated vertices:

```
sage: g=graphs.cospectral_graphs(6, graphs=lambda x: min(x.degree())>0)
sage: sorted(sorted(g.graph6_string() for g in glist) for glist in g)
[['Ep__', 'Er?G'], ['ExGg', 'ExoG']]
sage: g[0][1].am().charpoly()==g[0][1].am().charpoly()
True
sage: g[1][1].am().charpoly()==g[1][1].am().charpoly()
True
```

There is one pair of cospectral trees on eight vertices:

```
sage: g=graphs.cospectral_graphs(6, graphs=graphs.trees(8))
sage: sorted(sorted(g.graph6_string() for g in glist) for glist in g)
[['GiPC?C', 'GiQCC?']]
sage: g[0][1].am().charpoly()==g[0][1].am().charpoly()
True
```

There are two sets of cospectral graphs (with respect to the Laplacian matrix) on six vertices:

```
sage: g=graphs.cospectral_graphs(6, matrix_function=lambda g: g.laplacian_
↪matrix())
sage: sorted(sorted(g.graph6_string() for g in glist) for glist in g)
[['Edq_', 'ErcG'], ['Exoo', 'EzcG']]
sage: g[0][1].laplacian_matrix().charpoly()==g[0][1].laplacian_matrix().
↪charpoly()
True
sage: g[1][1].laplacian_matrix().charpoly()==g[1][1].laplacian_matrix().
↪charpoly()
True
```

To find cospectral graphs with respect to the normalized Laplacian, assuming the graphs do not have an isolated vertex, it is enough to check the spectrum of the matrix $D^{-1}A$, where D is the diagonal matrix of vertex degrees, and A is the adjacency matrix. We find two such cospectral graphs (for the normalized Laplacian) on five vertices:

```
sage: def DinverseA(g):
....:     A=g.adjacency_matrix().change_ring(QQ)
```

(continues on next page)

(continued from previous page)

```
....:     for i in range(g.order()):
....:         A.rescale_row(i, 1/len(A.nonzero_positions_in_row(i)))
....:     return A
sage: g=graphs.cospectral_graphs(5, matrix_function=DinverseA, graphs=lambda␣
↪g: min(g.degree())>0)
sage: sorted(sorted(g.graph6_string() for g in glist) for glist in g)
[['Dlg', 'Ds_']]
sage: g[0][1].laplacian_matrix(normalized=True).charpoly()==g[0][1].laplacian_
↪matrix(normalized=True).charpoly()
True
```

fullerenes(*order*, *ipr=False*)

Returns a generator which creates fullerene graphs using the buckygen generator (see [BGM2012]).

INPUT:

- `order` - a positive even integer smaller than or equal to 254. This specifies the number of vertices in the generated fullerenes.
- `ipr` - default: `False` - if `True` only fullerenes that satisfy the Isolated Pentagon Rule are generated. This means that no pentagonal faces share an edge.

OUTPUT:

A generator which will produce the fullerene graphs as Sage graphs with an embedding set. These will be simple graphs: no loops, no multiple edges, no directed edges.

See also:

- *set_embedding()*, *get_embedding()* – get/set methods for embeddings.

EXAMPLES:

There are 1812 isomers of C_{60}, i.e., 1812 fullerene graphs on 60 vertices:

```
sage: gen = graphs.fullerenes(60)  # optional buckygen
sage: len(list(gen))  # optional buckygen
1812
```

However, there is only one IPR fullerene graph on 60 vertices: the famous Buckminster Fullerene:

```
sage: gen = graphs.fullerenes(60, ipr=True)  # optional buckygen
sage: next(gen)  # optional buckygen
Graph on 60 vertices
sage: next(gen)  # optional buckygen
Traceback (most recent call last):
...
StopIteration
```

The unique fullerene graph on 20 vertices is isomorphic to the dodecahedron graph.

```
sage: gen = graphs.fullerenes(20)  # optional buckygen
sage: g = next(gen)  # optional buckygen
sage: g.is_isomorphic(graphs.DodecahedralGraph()) # optional buckygen
True
sage: g.get_embedding()  # optional buckygen
{1: [2, 3, 4],
 2: [1, 5, 6],
```

(continues on next page)

(continued from previous page)

```
 3: [1, 7, 8],
 4: [1, 9, 10],
 5: [2, 10, 11],
 6: [2, 12, 7],
 7: [3, 6, 13],
 8: [3, 14, 9],
 9: [4, 8, 15],
 10: [4, 16, 5],
 11: [5, 17, 12],
 12: [6, 11, 18],
 13: [7, 18, 14],
 14: [8, 13, 19],
 15: [9, 19, 16],
 16: [10, 15, 17],
 17: [11, 16, 20],
 18: [12, 20, 13],
 19: [14, 20, 15],
 20: [17, 19, 18]}
sage: g.plot3d(layout='spring')  # optional buckygen
Graphics3d Object
```

fusenes (*hexagon_count*, *benzenoids=False*)

Returns a generator which creates fusenes and benzenoids using the benzene generator (see [BCH2002]). Fusenes are planar polycyclic hydrocarbons with all bounded faces hexagons. Benzenoids are fusenes that are subgraphs of the hexagonal lattice.

INPUT:

- `hexagon_count` - a positive integer smaller than or equal to 30. This specifies the number of hexagons in the generated benzenoids.
- `benzenoids` - default: `False` - if `True` only benzenoids are generated.

OUTPUT:

A generator which will produce the fusenes as Sage graphs with an embedding set. These will be simple graphs: no loops, no multiple edges, no directed edges.

See also:

- *set_embedding()*, *get_embedding()* – get/set methods for embeddings.

EXAMPLES:

There is a unique fusene with 2 hexagons:

```
sage: gen = graphs.fusenes(2)  # optional benzene
sage: len(list(gen))  # optional benzene
1
```

This fusene is naphthalene ($C_{10}H_8$). In the fusene graph the H-atoms are not stored, so this is a graph on just 10 vertices:

```
sage: gen = graphs.fusenes(2)  # optional benzene
sage: next(gen)  # optional benzene
Graph on 10 vertices
sage: next(gen)  # optional benzene
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
StopIteration
```

There are 6505 benzenoids with 9 hexagons:

```
sage: gen = graphs.fusenes(9, benzenoids=True)  # optional benzene
sage: len(list(gen))  # optional benzene
6505
```

static line_graph_forbidden_subgraphs()

Returns the 9 forbidden subgraphs of a line graph.

See the Wikipedia article Line_graph for more information.

The graphs are returned in the ordering given by the Wikipedia drawing, read from left to right and from top to bottom.

EXAMPLES:

```
sage: graphs.line_graph_forbidden_subgraphs()
[Claw graph: Graph on 4 vertices,
Graph on 6 vertices,
Graph on 6 vertices,
Graph on 5 vertices,
Graph on 6 vertices,
Graph on 6 vertices,
Graph on 6 vertices,
Graph on 6 vertices,
Graph on 5 vertices]
```

nauty_geng(*options=''*, *debug=False*)

Return a generator which creates graphs from nauty's geng program.

INPUT:

- `options` – string (default: `""`); a string passed to `geng` as if it was run at a system command line. At a minimum, you *must* pass the number of vertices you desire. Sage expects the graphs to be in nauty's "graph6" format, do not set an option to change this default or results will be unpredictable.
- `debug` – boolean (default: `False`); if `True` the first line of `geng`'s output to standard error is captured and the first call to the generator's `next()` function will return this line as a string. A line leading with ">A" indicates a successful initiation of the program with some information on the arguments, while a line beginning with ">E" indicates an error with the input.

The possible options, obtained as output of `geng --help`:

```
     n     : the number of vertices
mine:maxe  : a range for the number of edges
             #:0 means '# or more' except in the case 0:0
  res/mod  : only generate subset res out of subsets 0..mod-1

    -c     : only write connected graphs
    -C     : only write biconnected graphs
    -t     : only generate triangle-free graphs
    -f     : only generate 4-cycle-free graphs
    -b     : only generate bipartite graphs
                  (-t, -f and -b can be used in any combination)
    -m     : save memory at the expense of time (only makes a
```

(continues on next page)

(continued from previous page)

```
                difference in the absence of -b, -t, -f and n <= 28).
   -d#   : a lower bound for the minimum degree
   -D#   : a upper bound for the maximum degree
   -v    : display counts by number of edges
   -l    : canonically label output graphs

   -q    : suppress auxiliary output (except from -v)
```

Options which cause `geng` to use an output format different than the graph6 format are not listed above (-u, -g, -s, -y, -h) as they will confuse the creation of a Sage graph. The res/mod option can be useful when using the output in a routine run several times in parallel.

OUTPUT:

A generator which will produce the graphs as Sage graphs. These will be simple graphs: no loops, no multiple edges, no directed edges.

See also:

Graph.is_strongly_regular() – tests whether a graph is strongly regular and/or returns its parameters.

EXAMPLES:

The generator can be used to construct graphs for testing, one at a time (usually inside a loop). Or it can be used to create an entire list all at once if there is sufficient memory to contain it.

```
sage: gen = graphs.nauty_geng("2")
sage: next(gen)
Graph on 2 vertices
sage: next(gen)
Graph on 2 vertices
sage: next(gen)
Traceback (most recent call last):
...
StopIteration
```

A list of all graphs on 7 vertices. This agrees with OEIS sequence A000088.

```
sage: gen = graphs.nauty_geng("7")
sage: len(list(gen))
1044
```

A list of just the connected graphs on 7 vertices. This agrees with OEIS sequence A001349.

```
sage: gen = graphs.nauty_geng("7 -c")
sage: len(list(gen))
853
```

The `debug` switch can be used to examine `geng`'s reaction to the input in the `options` string. We illustrate success. (A failure will be a string beginning with ">E".) Passing the "-q" switch to `geng` will supress the indicator of a successful initiation, and so the first returned value might be an empty string if `debug` is `True`:

```
sage: gen = graphs.nauty_geng("4", debug=True)
sage: print(next(gen))
>A ...geng -d0D3 n=4 e=0-6
sage: gen = graphs.nauty_geng("4 -q", debug=True)
```

(continues on next page)

(continued from previous page)

```
sage: next(gen)
''
```

static petersen_family(*generate=False*)

Returns the Petersen family

The Petersen family is a collection of 7 graphs which are the forbidden minors of the linklessly embeddable graphs. For more information see the Wikipedia article Petersen_family.

INPUT:

- `generate` (boolean) – whether to generate the family from the $\Delta - Y$ transformations. When set to `False` (default) a hardcoded version of the graphs (with a prettier layout) is returned.

EXAMPLES:

```
sage: graphs.petersen_family()
[Petersen graph: Graph on 10 vertices,
 Complete graph: Graph on 6 vertices,
 Multipartite Graph with set sizes [3, 3, 1]: Graph on 7 vertices,
 Graph on 8 vertices,
 Graph on 9 vertices,
 Graph on 7 vertices,
 Graph on 8 vertices]
```

The two different inputs generate the same graphs:

```
sage: F1 = graphs.petersen_family(generate=False)
sage: F2 = graphs.petersen_family(generate=True)
sage: F1 = [g.canonical_label().graph6_string() for g in F1]
sage: F2 = [g.canonical_label().graph6_string() for g in F2]
sage: set(F1) == set(F2)
True
```

planar_graphs(*order*, *minimum_degree=None*, *minimum_connectivity=None*, *exact_connectivity=False*, *only_bipartite=False*, *dual=False*)

An iterator over connected planar graphs using the plantri generator.

This uses the plantri generator (see [BM2007]) which is available through the optional package plantri.

Note: The non-3-connected graphs will be returned several times, with all its possible embeddings.

INPUT:

- `order` - a positive integer smaller than or equal to 64. This specifies the number of vertices in the generated graphs.
- `minimum_degree` - default: `None` - a value ≥ 1 and ≤ 5, or `None`. This specifies the minimum degree of the generated graphs. If this is `None` and the order is 1, then this is set to 0. If this is `None` and the minimum connectivity is specified, then this is set to the same value as the minimum connectivity. If the minimum connectivity is also equal to `None`, then this is set to 1.
- `minimum_connectivity` - default: `None` - a value ≥ 1 and ≤ 3, or `None`. This specifies the minimum connectivity of the generated graphs. If this is `None` and the minimum degree is specified, then this is set to the minimum of the minimum degree and 3. If the minimum degree is also equal to `None`, then this is set to 1.

- `exact_connectivity` - default: `False` - if `True` only graphs with exactly the specified connectivity will be generated. This option cannot be used with `minimum_connectivity=3`, or if the minimum connectivity is not explicitly set.
- `only_bipartite` - default: `False` - if `True` only bipartite graphs will be generated. This option cannot be used for graphs with a minimum degree larger than 3.
- `dual` - default: `False` - if `True` return instead the planar duals of the generated graphs.

OUTPUT:

An iterator which will produce all planar graphs with the given number of vertices as Sage graphs with an embedding set. These will be simple graphs (no loops, no multiple edges, no directed edges) unless the option `dual=True` is used.

See also:

- *set_embedding()*, *get_embedding()* – get/set methods for embeddings.

EXAMPLES:

There are 6 planar graphs on 4 vertices:

```
sage: gen = graphs.planar_graphs(4)  # optional plantri
sage: len(list(gen))  # optional plantri
6
```

Three of these planar graphs are bipartite:

```
sage: gen = graphs.planar_graphs(4, only_bipartite=True)  # optional plantri
sage: len(list(gen))  # optional plantri
3
```

Setting `dual=True` gives the planar dual graphs:

```
sage: gen = graphs.planar_graphs(4, dual=True)  # optional plantri
sage: [u for u in list(gen)]  # optional plantri
[Graph on 4 vertices,
Multi-graph on 3 vertices,
Multi-graph on 2 vertices,
Looped multi-graph on 2 vertices,
Looped multi-graph on 1 vertex,
Looped multi-graph on 1 vertex]
```

The cycle of length 4 is the only 2-connected bipartite planar graph on 4 vertices:

```
sage: l = list(graphs.planar_graphs(4, minimum_connectivity=2, only_
↪bipartite=True))  # optional plantri
sage: l[0].get_embedding()  # optional plantri
{1: [2, 3],
 2: [1, 4],
 3: [1, 4],
 4: [2, 3]}
```

There is one planar graph with one vertex. This graph obviously has minimum degree equal to 0:

```
sage: list(graphs.planar_graphs(1))  # optional plantri
[Graph on 1 vertex]
sage: list(graphs.planar_graphs(1, minimum_degree=1))  # optional plantri
[]
```

quadrangulations(*order*, *minimum_degree=None*, *minimum_connectivity=None*, *no_nonfacial_quadrangles=False*, *dual=False*)

An iterator over planar quadrangulations using the plantri generator.

This uses the plantri generator (see [BM2007]) which is available through the optional package plantri.

INPUT:

- `order` - a positive integer smaller than or equal to 64. This specifies the number of vertices in the generated quadrangulations.
- `minimum_degree` - default: `None` - a value ≥ 2 and ≤ 3, or `None`. This specifies the minimum degree of the generated quadrangulations. If this is `None` and the minimum connectivity is specified, then this is set to the same value as the minimum connectivity. If the minimum connectivity is also equal to `None`, then this is set to 2.
- `minimum_connectivity` - default: `None` - a value ≥ 2 and ≤ 3, or `None`. This specifies the minimum connectivity of the generated quadrangulations. If this is `None` and the option `no_nonfacial_quadrangles` is set to `True`, then this is set to 3. Otherwise if this is `None` and the minimum degree is specified, then this is set to the minimum degree. If the minimum degree is also equal to `None`, then this is set to 3.
- `no_nonfacial_quadrangles` - default: `False` - if `True` only quadrangulations with no non-facial quadrangles are generated. This option cannot be used if `minimum_connectivity` is set to 2.
- `dual` - default: `False` - if `True` return instead the planar duals of the generated graphs.

OUTPUT:

An iterator which will produce all planar quadrangulations with the given number of vertices as Sage graphs with an embedding set. These will be simple graphs (no loops, no multiple edges, no directed edges).

See also:

- *set_embedding()*, *get_embedding()* – get/set methods for embeddings.

EXAMPLES:

The cube is the only 3-connected planar quadrangulation on 8 vertices:

```
sage: gen = graphs.quadrangulations(8, minimum_connectivity=3)  # optional␣
↪plantri
sage: g = next(gen)                                             # optional␣
↪plantri
sage: g.is_isomorphic(graphs.CubeGraph(3))                      # optional␣
↪plantri
True
sage: next(gen)                                                 # optional␣
↪plantri
Traceback (most recent call last):
...
StopIteration
```

An overview of the number of quadrangulations on up to 12 vertices. This agrees with OEIS sequence A113201:

```
sage: for i in range(4,13):                                  # optional plantri
....:     L =  len(list(graphs.quadrangulations(i)))         # optional plantri
```

(continues on next page)

(continued from previous page)

```
....:     print("{:2d}    {:3d}".format(i,L))          # optional plantri
 4      1
 5      1
 6      2
 7      3
 8      9
 9     18
10     62
11    198
12    803
```

There are 2 planar quadrangulation on 12 vertices that do not have a non-facial quadrangle:

```
sage: len([g for g in graphs.quadrangulations(12, no_nonfacial_
↪quadrangles=True)])  # optional plantri
2
```

Setting `dual=True` gives the planar dual graphs:

```
sage: [len(g) for g in graphs.quadrangulations(12, no_nonfacial_
↪quadrangles=True, dual=True)]  # optional plantri
[10, 10]
```

static strongly_regular_graph(*v*, *k*, *l*, *mu=-1*, *existence=False*, *check=True*)

Return a (v, k, λ, μ)-strongly regular graph.

This function relies partly on Andries Brouwer's database of strongly regular graphs. See the documentation of *sage.graphs.strongly_regular_db* for more information.

INPUT:

- `v,k,l,mu` (integers) – note that `mu`, if unspecified, is automatically determined from `v,k,l`.
- `existence` (boolean;‘‘False‘‘) – instead of building the graph, return:
  - `True` – meaning that a (v, k, λ, μ)-strongly regular graph exists.
  - `Unknown` – meaning that Sage does not know if such a strongly regular graph exists (see `sage.misc.unknown`).
  - `False` – meaning that no such strongly regular graph exists.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

Petersen's graph from its set of parameters:

```
sage: graphs.strongly_regular_graph(10,3,0,1,existence=True)
True
sage: graphs.strongly_regular_graph(10,3,0,1)
complement(Johnson graph with parameters 5,2): Graph on 10 vertices
```

Now without specifying μ:

```
sage: graphs.strongly_regular_graph(10,3,0)
complement(Johnson graph with parameters 5,2): Graph on 10 vertices
```

An obviously infeasible set of parameters:

```
sage: graphs.strongly_regular_graph(5,5,5,5,existence=True)
False
sage: graphs.strongly_regular_graph(5,5,5,5)
Traceback (most recent call last):
...
ValueError: There exists no (5, 5, 5, 5)-strongly regular graph
```

An set of parameters proved in a paper to be infeasible:

```
sage: graphs.strongly_regular_graph(324,57,0,12,existence=True)
False
sage: graphs.strongly_regular_graph(324,57,0,12)
Traceback (most recent call last):
...
EmptySetError: Andries Brouwer's database reports that no (324, 57, 0,
12)-strongly regular graph exists. Comments: <a
href="srgtabrefs.html#GavrilyukMakhnev05">Gavrilyuk & Makhnev</a> ...
```

A set of parameters unknown to be realizable in Andries Brouwer's database:

```
sage: graphs.strongly_regular_graph(324,95,22,30,existence=True)
Unknown
sage: graphs.strongly_regular_graph(324,95,22,30)
Traceback (most recent call last):
...
RuntimeError: Andries Brouwer's database reports that no
(324, 95, 22, 30)-strongly regular graph is known to exist.
Comments:
```

A large unknown set of parameters (not in Andries Brouwer's database):

```
sage: graphs.strongly_regular_graph(1394,175,0,25,existence=True)
Unknown
sage: graphs.strongly_regular_graph(1394,175,0,25)
Traceback (most recent call last):
...
RuntimeError: Sage cannot figure out if a (1394, 175, 0, 25)-strongly
regular graph exists.
```

Test the Claw bound (see 3.D of [BL1984]):

```
sage: graphs.strongly_regular_graph(2058,242,91,20,existence=True)
False
```

static trees(*vertices*)

Returns a generator of the distinct trees on a fixed number of vertices.

INPUT:

- `vertices` - the size of the trees created.

OUTPUT:

A generator which creates an exhaustive, duplicate-free listing of the connected free (unlabeled) trees with `vertices` number of vertices. A tree is a graph with no cycles.

ALGORITHM:

Uses an algorithm that generates each new tree in constant time. See the documentation for, and implementation of, the *sage.graphs.trees* module, including a citation.

EXAMPLES:

We create an iterator, then loop over its elements.

```
sage: tree_iterator = graphs.trees(7)
sage: for T in tree_iterator:
....:     print(T.degree_sequence())
[2, 2, 2, 2, 2, 1, 1]
[3, 2, 2, 2, 1, 1, 1]
[3, 2, 2, 2, 1, 1, 1]
[4, 2, 2, 1, 1, 1, 1]
[3, 3, 2, 1, 1, 1, 1]
[3, 3, 2, 1, 1, 1, 1]
[4, 3, 1, 1, 1, 1, 1]
[3, 2, 2, 2, 1, 1, 1]
[4, 2, 2, 1, 1, 1, 1]
[5, 2, 1, 1, 1, 1, 1]
[6, 1, 1, 1, 1, 1, 1]
```

The number of trees on the first few vertex counts. This is sequence A000055 in Sloane's OEIS.

```
sage: [len(list(graphs.trees(i))) for i in range(0, 15)]
[1, 1, 1, 1, 2, 3, 6, 11, 23, 47, 106, 235, 551, 1301, 3159]
```

triangulations(*order*, *minimum_degree=None*, *minimum_connectivity=None*, *exact_connectivity=False*, *only_eulerian=False*, *dual=False*)

An iterator over connected planar triangulations using the plantri generator.

This uses the plantri generator (see [BM2007]) which is available through the optional package plantri.

INPUT:

- `order` - a positive integer smaller than or equal to 64. This specifies the number of vertices in the generated triangulations.
- `minimum_degree` - default: `None` - a value ≥ 3 and ≤ 5, or `None`. This specifies the minimum degree of the generated triangulations. If this is `None` and the minimum connectivity is specified, then this is set to the same value as the minimum connectivity. If the minimum connectivity is also equal to `None`, then this is set to 3.
- `minimum_connectivity` - default: `None` - a value ≥ 3 and ≤ 5, or `None`. This specifies the minimum connectivity of the generated triangulations. If this is `None` and the minimum degree is specified, then this is set to the minimum of the minimum degree and 3. If the minimum degree is also equal to `None`, then this is set to 3.
- `exact_connectivity` - default: `False` - if `True` only triangulations with exactly the specified connectivity will be generated. This option cannot be used with `minimum_connectivity=3`, or if the minimum connectivity is not explicitly set.
- `only_eulerian` - default: `False` - if `True` only Eulerian triangulations will be generated. This option cannot be used if the minimum degree is explicitly set to anything else than 4.
- `dual` - default: `False` - if `True` return instead the planar duals of the generated graphs.

OUTPUT:

An iterator which will produce all planar triangulations with the given number of vertices as Sage graphs with an embedding set. These will be simple graphs (no loops, no multiple edges, no directed edges).

See also:

- *set_embedding()*, *get_embedding()* – get/set methods for embeddings.
- *RandomTriangulation()* – build a random triangulation.

EXAMPLES:

The unique planar embedding of the K_4 is the only planar triangulations on 4 vertices:

```
sage: gen = graphs.triangulations(4)     # optional plantri
sage: [g.get_embedding() for g in gen]   # optional plantri
[{1: [2, 3, 4], 2: [1, 4, 3], 3: [1, 2, 4], 4: [1, 3, 2]}]
```

but, of course, this graph is not Eulerian:

```
sage: gen = graphs.triangulations(4, only_eulerian=True)  # optional plantri
sage: len(list(gen))                                      # optional plantri
0
```

The unique Eulerian triangulation on 6 vertices is isomorphic to the octahedral graph.

```
sage: gen = graphs.triangulations(6, only_eulerian=True)  # optional plantri
sage: g = next(gen)                                       # optional plantri
sage: g.is_isomorphic(graphs.OctahedralGraph())           # optional plantri
True
```

An overview of the number of 5-connected triangulations on up to 22 vertices. This agrees with OEIS sequence A081621:

```
sage: for i in range(12, 23):                                                #␣
↪optional plantri
....:     L = len(list(graphs.triangulations(i, minimum_connectivity=5))) #␣
↪optional plantri
....:     print("{}   {:3d}".format(i,L))                                   #␣
↪optional plantri
12     1
13     0
14     1
15     1
16     3
17     4
18    12
19    23
20    71
21   187
22   627
```

The minimum connectivity can be at most the minimum degree:

```
sage: gen = next(graphs.triangulations(10, minimum_degree=3, minimum_
↪connectivity=5))  # optional plantri
Traceback (most recent call last):
...
ValueError: Minimum connectivity can be at most the minimum degree.
```

There are 5 triangulations with 9 vertices and minimum degree equal to 4 that are 3-connected, but only one of them is not 4-connected:

```
sage: len([g for g in graphs.triangulations(9, minimum_degree=4, minimum_
↪connectivity=3)]) # optional plantri
5
sage: len([g for g in graphs.triangulations(9, minimum_degree=4, minimum_
↪connectivity=3, exact_connectivity=True)]) # optional plantri
1
```

Setting `dual=True` gives the planar dual graphs:

```
sage: [len(g) for g in graphs.triangulations(9, minimum_degree=4, minimum_
↪connectivity=3, dual=True)]  # optional plantri
[14, 14, 14, 14, 14]
```

sage.graphs.graph_generators.**canaug_traverse_edge**(*g*, *aut_gens*, *property*, *dig=False*, *loops=False*, *sparse=True*)

Main function for exhaustive generation. Recursive traversal of a canonically generated tree of isomorph free graphs satisfying a given property.

INPUT:

- `g` - current position on the tree.
- `aut_gens` - list of generators of Aut(g), in list notation.
- `property` - check before traversing below g.

EXAMPLES:

```
sage: from sage.graphs.graph_generators import canaug_traverse_edge
sage: G = Graph(3)
sage: list(canaug_traverse_edge(G, [], lambda x: True))
[Graph on 3 vertices, ... Graph on 3 vertices]
```

The best way to access this function is through the graphs() iterator:

Print graphs on 3 or less vertices.

```
sage: for G in graphs(3):
....:     print(G)
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
```

Print digraphs on 3 or less vertices.

```
sage: for G in digraphs(3):
....:     print(G)
Digraph on 3 vertices
Digraph on 3 vertices
...
Digraph on 3 vertices
Digraph on 3 vertices
```

sage.graphs.graph_generators.**canaug_traverse_vert**(*g*, *aut_gens*, *max_verts*, *property*, *dig=False*, *loops=False*, *sparse=True*)

Main function for exhaustive generation. Recursive traversal of a canonically generated tree of isomorph free (di)graphs satisfying a given property.

INPUT:

- `g` - current position on the tree.
- `aut_gens` - list of generators of Aut(g), in list notation.
- `max_verts` - when to retreat.
- `property` - check before traversing below g.
- `degree_sequence` - specify a degree sequence to try to obtain.

EXAMPLES:

```
sage: from sage.graphs.graph_generators import canaug_traverse_vert
sage: list(canaug_traverse_vert(Graph(), [], 3, lambda x: True))
[Graph on 0 vertices, ... Graph on 3 vertices]
```

The best way to access this function is through the graphs() iterator:

Print graphs on 3 or less vertices.

```
sage: for G in graphs(3, augment='vertices'):
....:     print(G)
Graph on 0 vertices
Graph on 1 vertex
Graph on 2 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 2 vertices
Graph on 3 vertices
```

Print digraphs on 2 or less vertices.

```
sage: for D in digraphs(2, augment='vertices'):
....:     print(D)
Digraph on 0 vertices
Digraph on 1 vertex
Digraph on 2 vertices
Digraph on 2 vertices
Digraph on 2 vertices
```

sage.graphs.graph_generators.**check_aut**(*aut_gens*, *cut_vert*, *n*)

Helper function for exhaustive generation.

At the start, check_aut is given a set of generators for the automorphism group, aut_gens. We already know we are looking for an element of the auto- morphism group that sends cut_vert to n, and check_aut generates these for the canaug_traverse function.

EXAMPLES:

Note that the last two entries indicate that none of the automorphism group has yet been searched - we are starting at the identity [0, 1, 2, 3] and so far that is all we have seen. We return automorphisms mapping 2 to 3:

```
sage: from sage.graphs.graph_generators import check_aut
sage: list( check_aut( [ [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3] ], 2, 3))
[[1, 0, 3, 2], [1, 2, 3, 0]]
```

sage.graphs.graph_generators.**check_aut_edge**(*aut_gens*, *cut_edge*, *i*, *j*, *n*, *dig=False*)

Helper function for exhaustive generation.

At the start, check_aut_edge is given a set of generators for the automorphism group, aut_gens. We already know we are looking for an element of the auto- morphism group that sends cut_edge to {i, j}, and check_aut generates these for the canaug_traverse function.

EXAMPLES:

Note that the last two entries indicate that none of the automorphism group has yet been searched - we are starting at the identity [0, 1, 2, 3] and so far that is all we have seen. We return automorphisms mapping 2 to 3:

```
sage: from sage.graphs.graph_generators import check_aut
sage: list( check_aut( [ [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3] ], 2, 3))
[[1, 0, 3, 2], [1, 2, 3, 0]]
```

2.2 Common Digraphs

All digraphs in Sage can be built through the `digraphs` object. In order to build a circuit on 15 elements, one can do:

```
sage: g = digraphs.Circuit(15)
```

To get a circulant graph on 10 vertices in which a vertex i has $i + 2$ and $i + 3$ as outneighbors:

```
sage: p = digraphs.Circulant(10,[2,3])
```

More interestingly, one can get the list of all digraphs that Sage knows how to build by typing `digraphs.` in Sage and then hitting tab.

ButterflyGraph()	Return a n-dimensional butterfly graph.
Circuit()	Return the circuit on n vertices.
Circulant()	Return a circulant digraph on n vertices from a set of integers.
Complete()	Return a complete digraph on n vertices.
DeBruijn()	Return the De Bruijn digraph with parameters k, n.
GeneralizedDeBruijn()	Return the generalized de Bruijn digraph of order n and degree d.
ImaseItoh()	Return the digraph of Imase and Itoh of order n and degree d.
Kautz()	Return the Kautz digraph of degree d and diameter D.
nauty_directg()	Return an iterator yielding digraphs using nauty's `directg` program.
Paley()	Return a Paley digraph on q vertices.
Path()	Return a directed path on n vertices.
RandomDirectedGNC()	Return a random growing network with copying (GNC) digraph with n vertices.
RandomDirectedGNM()	Return a random labelled digraph on n nodes and m arcs.
RandomDirectedGNP()	Return a random digraph on n nodes.
RandomDirectedGN()	Return a random growing network (GN) digraph with n vertices.
RandomDirectedGNR()	Return a random growing network with redirection (GNR) digraph.
RandomSemiComplete()	Return a random semi-complete digraph of order n.
RandomTournament()	Return a random tournament on n vertices.
TransitiveTournament()	Return a transitive tournament on n vertices.
tournaments_nauty()	Iterator over all tournaments on n vertices using Nauty.

AUTHORS:

- Robert L. Miller (2006)
- Emily A. Kirkman (2006)

- Michael C. Yurko (2009)
- David Coudert (2012)

2.2.1 Functions and methods

class sage.graphs.digraph_generators.**DiGraphGenerators**
Bases: object

A class consisting of constructors for several common digraphs, including orderly generation of isomorphism class representatives.

A list of all graphs and graph structures in this database is available via tab completion. Type "digraphs." and then hit tab to see which graphs are available.

The docstrings include educational information about each named digraph with the hopes that this class can be used as a reference.

The constructors currently in this class include:

```
Random Directed Graphs:
    - RandomDirectedGN
    - RandomDirectedGNC
    - RandomDirectedGNP
    - RandomDirectedGNM
    - RandomDirectedGNR
    - RandomTournament
    - RandomSemiComplete

Families of Graphs:
    - Complete
    - DeBruijn
    - GeneralizedDeBruijn
    - Kautz
    - Path
    - ImaseItoh
    - RandomTournament
    - TransitiveTournament
    - tournaments_nauty
```

ORDERLY GENERATION: digraphs(vertices, property=lambda x: True, augment='edges', size=None)

Accesses the generator of isomorphism class representatives [McK1998]. Iterates over distinct, exhaustive representatives.

INPUT:

- `vertices` – natural number or `None` to infinitely generate bigger and bigger digraphs.
- `property` – any property to be tested on digraphs before generation
- `augment` – choices:
  - `'vertices'` – augments by adding a vertex, and edges incident to that vertex. In this case, all digraphs on *up to* n=vertices are generated. If for any digraph G satisfying the property, every subgraph, obtained from G by deleting one vertex and only edges incident to that vertex, satisfies the property, then this will generate all digraphs with that property. If this does not hold, then all the digraphs generated will satisfy the property, but there will be some missing.
  - `'edges'` – augments a fixed number of vertices by adding one edge. In this case, all digraphs on *exactly* n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained

from G by deleting one edge but not the vertices incident to that edge, satisfies the property, then this will generate all digraphs with that property. If this does not hold, then all the digraphs generated will satisfy the property, but there will be some missing.

- `implementation` – which underlying implementation to use (see DiGraph?)
- `sparse` – boolean (default: `True`); whether to use a sparse or dense data structure. See the documentation of *Graph*.

EXAMPLES:

Print digraphs on 2 or less vertices:

```
sage: for D in digraphs(2, augment='vertices'):
....:     print(D)
Digraph on 0 vertices
Digraph on 1 vertex
Digraph on 2 vertices
Digraph on 2 vertices
Digraph on 2 vertices
```

Print digraphs on 3 vertices:

```
sage: for D in digraphs(3):
....:     print(D)
Digraph on 3 vertices
Digraph on 3 vertices
...
Digraph on 3 vertices
Digraph on 3 vertices
```

Generate all digraphs with 4 vertices and 3 edges:

```
sage: L = digraphs(4, size=3)
sage: len(list(L))
13
```

Generate all digraphs with 4 vertices and up to 3 edges:

```
sage: L = list(digraphs(4, lambda G: G.size() <= 3))
sage: len(L)
20
sage: graphs_list.show_graphs(L)  # long time
```

Generate all digraphs with degree at most 2, up to 5 vertices:

```
sage: property = lambda G: (max([G.degree(v) for v in G] + [0]) <= 2)
sage: L = list(digraphs(5, property, augment='vertices'))
sage: len(L)
75
```

Generate digraphs on the fly (see http://oeis.org/classic/A000273):

```
sage: for i in range(5):
....:     print(len(list(digraphs(i))))
1
1
3
16
218
```

ButterflyGraph(*n*, *vertices='strings'*)

Return a n-dimensional butterfly graph.

The vertices consist of pairs (v, i), where v is an n-dimensional tuple (vector) with binary entries (or a string representation of such) and i is an integer in $[0..n]$. A directed edge goes from (v, i) to $(w, i + 1)$ if v and w are identical except for possibly when $v[i] \neq w[i]$.

A butterfly graph has $(2^n)(n + 1)$ vertices and $n2^{n+1}$ edges.

INPUT:

- `n` – integer;
- `vertices` – string (default: `'strings'`); specifies whether the vertices are zero-one strings (default) or tuples over GF(2) (`vertices='vectors'`)

EXAMPLES:

```
sage: digraphs.ButterflyGraph(2).edges(labels=False)
[(('00', 0), ('00', 1)),
(('00', 0), ('10', 1)),
(('00', 1), ('00', 2)),
(('00', 1), ('01', 2)),
(('01', 0), ('01', 1)),
(('01', 0), ('11', 1)),
(('01', 1), ('00', 2)),
(('01', 1), ('01', 2)),
(('10', 0), ('00', 1)),
(('10', 0), ('10', 1)),
(('10', 1), ('10', 2)),
(('10', 1), ('11', 2)),
(('11', 0), ('01', 1)),
(('11', 0), ('11', 1)),
(('11', 1), ('10', 2)),
(('11', 1), ('11', 2))]
sage: digraphs.ButterflyGraph(2,vertices='vectors').edges(labels=False)
[(((0, 0), 0), ((0, 0), 1)),
(((0, 0), 0), ((1, 0), 1)),
(((0, 0), 1), ((0, 0), 2)),
(((0, 0), 1), ((0, 1), 2)),
(((0, 1), 0), ((0, 1), 1)),
(((0, 1), 0), ((1, 1), 1)),
(((0, 1), 1), ((0, 0), 2)),
(((0, 1), 1), ((0, 1), 2)),
(((1, 0), 0), ((0, 0), 1)),
(((1, 0), 0), ((1, 0), 1)),
(((1, 0), 1), ((1, 0), 2)),
(((1, 0), 1), ((1, 1), 2)),
(((1, 1), 0), ((0, 1), 1)),
(((1, 1), 0), ((1, 1), 1)),
(((1, 1), 1), ((1, 0), 2)),
(((1, 1), 1), ((1, 1), 2))]
```

Circuit(*n*)

Return the circuit on n vertices.

The circuit is an oriented `CycleGraph`.

EXAMPLES:

A circuit is the smallest strongly connected digraph:

```
sage: circuit = digraphs.Circuit(15)
sage: len(circuit.strongly_connected_components()) == 1
True
```

Circulant(*n*, *integers*)

Return a circulant digraph on n vertices from a set of integers.

INPUT:

- `n` – integer; number of vertices
- `integers` – iterable container (list, set, etc.) of integers such that there is an edge from i to j if and only if `(j-i)%n in integers`

EXAMPLES:

```
sage: digraphs.Circulant(13,[3,5,7])
Circulant graph ([3, 5, 7]): Digraph on 13 vertices
```

Complete(*n*, *loops=False*)

Return the complete digraph on n vertices.

INPUT:

- `n` – integer; number of vertices
- `loops` – boolean (default: `False`); whether to add loops or not, i.e., edges from u to itself

See also:

- *RandomSemiComplete()*
- *RandomTournament()*

EXAMPLES:

```
sage: n = 10
sage: G = digraphs.Complete(n); G
Complete digraph: Digraph on 10 vertices
sage: G.size() == n*(n-1)
True
sage: G = digraphs.Complete(n, loops=True); G
Complete digraph with loops: Looped digraph on 10 vertices
sage: G.size() == n*n
True
sage: digraphs.Complete(-1)
Traceback (most recent call last):
...
ValueError: the number of vertices cannot be strictly negative
```

DeBruijn(*k*, *n*, *vertices='strings'*)

Return the De Bruijn digraph with parameters k, n.

The De Bruijn digraph with parameters k, n is built upon a set of vertices equal to the set of words of length n from a dictionary of k letters.

In this digraph, there is an arc w_1w_2 if w_2 can be obtained from w_1 by removing the leftmost letter and adding a new letter at its right end. For more information, see the Wikipedia article De_Bruijn_graph.

INPUT:

- **k – two possibilities for this parameter :**

  - An integer equal to the cardinality of the alphabet to use, that is, the degree of the digraph to be produced.
  - An iterable object to be used as the set of letters. The degree of the resulting digraph is the cardinality of the set of letters.
- `n` – integer; length of words in the De Bruijn digraph when `vertices == 'strings'`, and also the diameter of the digraph.
- `vertices` – string (default: `'strings'`); whether the vertices are words over an alphabet (default) or integers (`vertices='string'`)

EXAMPLES:

de Bruijn digraph of degree 2 and diameter 2:

```
sage: db = digraphs.DeBruijn(2, 2); db
De Bruijn digraph (k=2, n=2): Looped digraph on 4 vertices
sage: db.order(), db.size()
(4, 8)
sage: db.diameter()
2
```

Building a de Bruijn digraph on a different alphabet:

```
sage: g = digraphs.DeBruijn(['a', 'b'], 2)
sage: g.vertices()
['aa', 'ab', 'ba', 'bb']
sage: g.is_isomorphic(db)
True
sage: g = digraphs.DeBruijn(['AA', 'BB'], 2)
sage: g.vertices()
['AA,AA', 'AA,BB', 'BB,AA', 'BB,BB']
sage: g.is_isomorphic(db)
True
```

GeneralizedDeBruijn(n, d)

Return the generalized de Bruijn digraph of order n and degree d.

The generalized de Bruijn digraph was defined in [RPK1980] [RPK1983]. It has vertex set $V = \{0, 1, ..., n-1\}$ and there is an arc from vertex $u \in V$ to all vertices $v \in V$ such that $v \equiv (u * d + a) \mod n$ with $0 \leq a < d$.

When $n = d^D$, the generalized de Bruijn digraph is isomorphic to the de Bruijn digraph of degree d and diameter D.

INPUT:

- `n` – integer; number of vertices of the digraph (must be at least one)
- `d` – integer; degree of the digraph (must be at least one)

See also:

- *sage.graphs.generic_graph.GenericGraph.is_circulant()* – checks whether a (di)graph is circulant, and/or returns all possible sets of parameters.

EXAMPLES:

```
sage: GB = digraphs.GeneralizedDeBruijn(8, 2)
sage: GB.is_isomorphic(digraphs.DeBruijn(2, 3), certificate = True)
(True, {0: '000', 1: '001', 2: '010', 3: '011', 4: '100', 5: '101', 6: '110',␣
↪7: '111'})
```

ImaseItoh(*n*, *d*)

Return the Imase-Itoh digraph of order n and degree d.

The Imase-Itoh digraph was defined in [II1983]. It has vertex set $V = \{0, 1, ..., n-1\}$ and there is an arc from vertex $u \in V$ to all vertices $v \in V$ such that $v \equiv (-u * d - a - 1) \mod n$ with $0 \leq a < d$.

When $n = d^D$, the Imase-Itoh digraph is isomorphic to the de Bruijn digraph of degree d and diameter D. When $n = d^{D-1}(d+1)$, the Imase-Itoh digraph is isomorphic to the Kautz digraph [Kau1968] of degree d and diameter D.

INPUT:

- `n` – integer; number of vertices of the digraph (must be greater than or equal to two)
- `d` – integer; degree of the digraph (must be greater than or equal to one)

EXAMPLES:

```
sage: II = digraphs.ImaseItoh(8, 2)
sage: II.is_isomorphic(digraphs.DeBruijn(2, 3), certificate = True)
(True, {0: '010', 1: '011', 2: '000', 3: '001', 4: '110', 5: '111', 6: '100',␣
↪7: '101'})

sage: II = digraphs.ImaseItoh(12, 2)
sage: b,D = II.is_isomorphic(digraphs.Kautz(2, 3), certificate=True)
sage: b
True
sage: D   # random isomorphism
{0: '202', 1: '201', 2: '210', 3: '212', 4: '121',
 5: '120', 6: '102', 7: '101', 8: '010', 9: '012',
 10: '021', 11: '020'}
```

Kautz(*k*, *D*, *vertices='strings'*)

Return the Kautz digraph of degree d and diameter D.

The Kautz digraph has been defined in [Kau1968]. The Kautz digraph of degree d and diameter D has $d^{D-1}(d+1)$ vertices. This digraph is built from a set of vertices equal to the set of words of length D over an alphabet of $d+1$ letters such that consecutive letters are different. There is an arc from vertex u to vertex v if v can be obtained from u by removing the leftmost letter and adding a new letter, distinct from the rightmost letter of u, at the right end.

The Kautz digraph of degree d and diameter D is isomorphic to the Imase-Itoh digraph [II1983] of degree d and order $d^{D-1}(d+1)$.

See the Wikipedia article Kautz_graph for more information.

INPUT:

- `k` – two possibilities for this parameter. In either case the degree must be at least one:
  - An integer equal to the degree of the digraph to be produced, that is, the cardinality of the alphabet to be used minus one.
  - An iterable object to be used as the set of letters. The degree of the resulting digraph is the cardinality of the set of letters minus one.

- `D` – integer; diameter of the digraph, and length of a vertex label when `vertices == 'strings'` (must be at least one)
- `vertices` – string (default: `'strings'`); whether the vertices are words over an alphabet (default) or integers (`vertices='strings'`)

EXAMPLES:

```
sage: K = digraphs.Kautz(2, 3)
sage: b,D = K.is_isomorphic(digraphs.ImaseItoh(12, 2), certificate=True)
sage: b
True
sage: D  # random isomorphism
{'010': 8, '012': 9, '020': 11, '021': 10, '101': 7,  '102': 6,
 '120': 5, '121': 4, '201': 1, '202': 0, '210': 2, '212': 3}

sage: K = digraphs.Kautz([1,'a','B'], 2)
sage: K.edges()
[('1B', 'B1', '1'), ('1B', 'Ba', 'a'), ('1a', 'a1', '1'), ('1a', 'aB', 'B'), (
↪'B1', '1B', 'B'), ('B1', '1a', 'a'), ('Ba', 'a1', '1'), ('Ba', 'aB', 'B'), (
↪'a1', '1B', 'B'), ('a1', '1a', 'a'), ('aB', 'B1', '1'), ('aB', 'Ba', 'a')]

sage: K = digraphs.Kautz([1,'aA','BB'], 2)
sage: K.edges()
[('1,BB', 'BB,1', '1'), ('1,BB', 'BB,aA', 'aA'), ('1,aA', 'aA,1', '1'), ('1,aA
↪', 'aA,BB', 'BB'), ('BB,1', '1,BB', 'BB'), ('BB,1', '1,aA', 'aA'), ('BB,aA',
↪ 'aA,1', '1'), ('BB,aA', 'aA,BB', 'BB'), ('aA,1', '1,BB', 'BB'), ('aA,1',
↪'1,aA', 'aA'), ('aA,BB', 'BB,1', '1'), ('aA,BB', 'BB,aA', 'aA')]
```

Paley(*q*)

Return a Paley digraph on q vertices.

Parameter q must be the power of a prime number and congruent to 3 mod 4.

See also:

- Wikipedia article Paley_graph
- *PaleyGraph()*

EXAMPLES:

A Paley digraph has $n * (n-1)/2$ edges, its underlying graph is a clique, and so it is a tournament:

```
sage: g = digraphs.Paley(7); g
Paley digraph with parameter 7: Digraph on 7 vertices
sage: g.size() == g.order() * (g.order() - 1) / 2
True
sage: g.to_undirected().is_clique()
True
```

A Paley digraph is always self-complementary:

```
sage: g.complement().is_isomorphic(g)
True
```

Path(*n*)

Return a directed path on n vertices.

INPUT:

- n – integer; number of vertices in the path

EXAMPLES:

```
sage: g = digraphs.Path(5)
sage: g.vertices()
[0, 1, 2, 3, 4]
sage: g.size()
4
sage: g.automorphism_group().cardinality()
1
```

RandomDirectedGN(*n*, *kernel=<function DiGraphGenerators.<lambda> at 0x7f7e4e9ca620>*, *seed=None*)

Return a random growing network (GN) digraph with n vertices.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen with a preferential attachment model, i.e. probability is proportional to degree. The default attachment kernel is a linear function of degree. The digraph is always a tree, so in particular it is a directed acyclic graph. See [KR2001b] for more details.

INPUT:

- n – integer; number of vertices
- kernel – the attachment kernel
- seed – a random.Random seed or a Python int for the random number generator (default: None)

EXAMPLES:

```
sage: D = digraphs.RandomDirectedGN(25)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (3, 2), (4, 2), (5, 4), (6, 3), (7, 0), (8, 4), (9, 4), (10,␣
↪3), (11, 4), (12, 4), (13, 3), (14, 4), (15, 4), (16, 0), (17, 2), (18, 4),␣
↪(19, 6), (20, 14), (21, 4), (22, 0), (23, 22), (24, 14)]  # 32-bit
[(1, 0), (2, 1), (3, 0), (4, 2), (5, 0), (6, 2), (7, 3), (8, 2), (9, 3), (10,␣
↪4), (11, 5), (12, 9), (13, 2), (14, 2), (15, 5), (16, 2), (17, 15), (18, 1),
↪ (19, 5), (20, 2), (21, 5), (22, 1), (23, 5), (24, 14)]   # 64-bit
sage: D.show()  # long time
```

RandomDirectedGNC(*n*, *seed=None*)

Return a random growing network with copying (GNC) digraph with n vertices.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen with a preferential attachment model, i.e. probability is proportional to degree. The new vertex is also linked to all of the previously added vertex's successors. See [KR2005] for more details.

INPUT:

- n – integer; number of vertices
- seed – a random.Random seed or a Python int for the random number generator (default: None)

EXAMPLES:

```
sage: D = digraphs.RandomDirectedGNC(25)
sage: D.is_directed_acyclic()
True
sage: D.topological_sort()
[24, 23, ..., 1, 0]
sage: D.show()  # long time
```

RandomDirectedGNM(*n*, *m*, *loops=False*)

Return a random labelled digraph on n nodes and m arcs.

INPUT:

- `n` – integer; number of vertices
- `m` – integer; number of edges
- `loops` – boolean (default: `False`); whether to allow loops

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES:

```
sage: D = digraphs.RandomDirectedGNM(10, 5)
sage: D.num_verts()
10
sage: D.edges(labels=False)
[(0, 3), (1, 5), (5, 1), (7, 0), (8, 5)]
```

With loops:

```
sage: D = digraphs.RandomDirectedGNM(10, 100, loops = True)
sage: D.num_verts()
10
sage: D.loops()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None), (4, 4, None), (5, 5,␣
↪None), (6, 6, None), (7, 7, None), (8, 8, None), (9, 9, None)]
```

RandomDirectedGNP(*n*, *p*, *loops=False*, *seed=None*)

Return a random digraph on n nodes.

Each edge is inserted independently with probability p. See [ER1959] and [Gil1959] for more details.

INPUT:

- `n` – integer; number of nodes of the digraph
- `p` – float; probability of an edge
- `loops` – boolean (default: `False`); whether the random digraph may have loops
- `seed` – integer (default: `None`); seed for random number generator

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES:

```
sage: set_random_seed(0)
sage: D = digraphs.RandomDirectedGNP(10, .2)
sage: D.num_verts()
10
sage: D.edges(labels=False)
[(1, 0), (1, 2), (3, 6), (3, 7), (4, 5), (4, 7), (4, 8), (5, 2), (6, 0), (7,␣
↪2), (8, 1), (8, 9), (9, 4)]
```

RandomDirectedGNR(*n*, *p*, *seed=None*)

Return a random growing network with redirection (GNR) digraph with n vertices and redirection probability p.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen uniformly. With probability p, the arc is instead redirected to the successor vertex. The digraph is always a tree. See [KR2001b] for more details.

INPUT:

- `n` – integer; number of vertices
- `p` – redirection probability
- `seed` – a `random.Random` seed or a Python `int` for the random number generator (default: `None`)

EXAMPLES:

```
sage: D = digraphs.RandomDirectedGNR(25, .2)
sage: D.is_directed_acyclic()
True
sage: D.to_undirected().is_tree()
True
sage: D.show()  # long time
```

RandomSemiComplete(*n*)

Return a random semi-complete digraph on n vertices.

A directed graph $G = (V, E)$ is *semi-complete* if for any pair of vertices u and v, there is *at least* one arc between them.

To generate randomly a semi-complete digraph, we have to ensure, for any pair of distinct vertices u and v, that with probability $1/3$ we have only arc uv, with probability $1/3$ we have only arc vu, and with probability $1/3$ we have both arc uv and arc vu. We do so by selecting a random integer $coin$ in $[1, 3]$. When $coin == 1$ we select only arc uv, when $coin == 3$ we select only arc vu, and when $coin == 2$ we select both arcs. In other words, we select arc uv when $coin \leq 2$ and arc vu when $coin \geq 2$.

INPUT:

- `n` – integer; the number of nodes

See also:

- *Complete()*
- *RandomTournament()*

EXAMPLES:

```
sage: SC = digraphs.RandomSemiComplete(10); SC
Random Semi-Complete digraph: Digraph on 10 vertices
sage: SC.size() >= binomial(10, 2)
True
sage: digraphs.RandomSemiComplete(-1)
Traceback (most recent call last):
...
ValueError: the number of vertices cannot be strictly negative
```

RandomTournament(*n*)

Return a random tournament on n vertices.

For every pair of vertices, the tournament has an edge from i to j with probability $1/2$, otherwise it has an edge from j to i.

INPUT:

- n – integer; number of vertices

EXAMPLES:

```
sage: T = digraphs.RandomTournament(10); T
Random Tournament: Digraph on 10 vertices
sage: T.size() == binomial(10, 2)
True
sage: T.is_tournament()
True
sage: digraphs.RandomTournament(-1)
Traceback (most recent call last):
...
ValueError: the number of vertices cannot be strictly negative
```

See also:

- Wikipedia article Tournament_(graph_theory)
- *is_tournament()*
- *TransitiveTournament()*
- *Complete()*
- *RandomSemiComplete()*

TransitiveTournament(*n*)

Return a transitive tournament on n vertices.

In this tournament there is an edge from i to j if $i < j$.

See the Wikipedia article Tournament_(graph_theory) for more information.

INPUT:

- n – integer; number of vertices in the tournament

EXAMPLES:

```
sage: g = digraphs.TransitiveTournament(5)
sage: g.vertices()
[0, 1, 2, 3, 4]
sage: g.size()
10
sage: g.automorphism_group().cardinality()
1
```

See also:

- Wikipedia article Tournament_(graph_theory)
- *is_tournament()*
- *is_transitive()*
- *RandomTournament()*

nauty_directg(*graphs*, *options=''*, *debug=False*)

Return an iterator yielding digraphs using nauty's `directg` program.

Description from directg –help: Read undirected graphs and orient their edges in all possible ways. Edges can be oriented in either or both directions (3 possibilities). Isomorphic directed graphs derived from the same input are suppressed. If the input graphs are non-isomorphic then the output graphs are also.

INPUT:

- `graphs` – a *Graph* or an iterable containing *Graph* the graph6 string of these graphs is used as an input for `directg`.
- `options` (str) – a string passed to directg as if it was run at a system command line. Available options from directg –help:

```
-e# | -e#:#  specify a value or range of the total number of arcs
-o     orient each edge in only one direction, never both
-f#  Use only the subgroup that fixes the first # vertices setwise
-V  only output graphs with nontrivial groups (including exchange of
      isolated vertices).  The -f option is respected.
-s#/#  Make only a fraction of the orientations: The first integer is
        the part number (first is 0) and the second is the number of
        parts. Splitting is done per input graph independently.
```

- `debug` (boolean) – default: `False` - if `True` directg standard error and standard output are displayed.

EXAMPLES:

```
sage: gen = graphs.nauty_geng("-c 3")
sage: dgs = list(digraphs.nauty_directg(gen))
sage: len(dgs)
13
sage: dgs[0]
Digraph on 3 vertices
sage: dgs[0]._bit_vector()
'001001000'
sage: len(list(digraphs.nauty_directg(graphs.PetersenGraph(), options="-o")))
324
```

See also:

- *orientations()*

tournaments_nauty(*n*, *min_out_degree=None*, *max_out_degree=None*, *strongly_connected=False*, *debug=False*, *options=''*)

Iterator over all tournaments on n vertices using Nauty.

INPUT:

- `n` – integer; number of vertices
- `min_out_degree`, `max_out_degree` – integers; if set to `None` (default), then the min/max out-degree is not constrained
- `debug` – boolean (default: `False`); if `True` the first line of genbg's output to standard error is captured and the first call to the generator's `next()` function will return this line as a string. A line leading with ">A" indicates a successful initiation of the program with some information on the arguments, while a line beginning with ">E" indicates an error with the input.
- `options` – string; anything else that should be forwarded as input to Nauty's genbg. See its documentation for more information : http://cs.anu.edu.au/~bdm/nauty/.

EXAMPLES:

```
sage: for g in digraphs.tournaments_nauty(4):
....:     print(g.edges(labels = False))
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
[(1, 0), (1, 3), (2, 0), (2, 1), (3, 0), (3, 2)]
[(0, 2), (1, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
[(0, 2), (0, 3), (1, 0), (2, 1), (3, 1), (3, 2)]
sage: tournaments = digraphs.tournaments_nauty
sage: [len(list(tournaments(x))) for x in range(1,8)]
[1, 1, 2, 4, 12, 56, 456]
sage: [len(list(tournaments(x, strongly_connected = True))) for x in range(1,
↪9)]
[1, 0, 1, 1, 6, 35, 353, 6008]
```

2.3 Common graphs and digraphs generators (Cython)

AUTHORS:

- David Coudert (2012)

sage.graphs.graph_generators_pyx.**RandomGNP**(*n*, *p*, *directed=False*, *loops=False*)

Return a random graph or a digraph on n nodes.

Each edge is inserted independently with probability p.

INPUT:

- `n` – number of nodes of the digraph
- `p` – probability of an edge
- `directed` – boolean (default: `False`); whether the random graph is directed or undirected (default)
- `loops` – boolean (default: `False`); whether the random digraph may have loops or not. This value is used only when `directed == True`.

REFERENCES:

- [ER1959]
- [Gil1959]

EXAMPLES:

```
sage: from sage.graphs.graph_generators_pyx import RandomGNP
sage: set_random_seed(0)
sage: D = RandomGNP(10, .2, directed=True)
sage: D.num_verts()
10
sage: D.edges(labels=False)
[(0, 2), (0, 5), (1, 5), (1, 7), (4, 1), (4, 2), (4, 9), (5, 0), (5, 2), (5, 3),␣
↪(5, 7), (6, 5), (7, 1), (8, 2), (8, 6), (9, 4)]
```

2.4 Graph database

This module implements classes (*GraphDatabase*, *GraphQuery*, *GenericGraphQuery*) for interfacing with the `sqlite` database `graphs.db`.

The *GraphDatabase* class interfaces with the sqlite database graphs.db. It is an immutable database that inherits from SQLDatabase (see sage.databases.sql_db).

The database contains all unlabeled graphs with 7 or fewer nodes. This class will also interface with the optional database package containing all unlabeled graphs with 8 or fewer nodes. The database(s) consists of five tables, and has the structure given by the function *graph_db_info()* (For a full description including column data types, create a *GraphDatabase* instance and call the method get_skeleton()).

AUTHORS:

- Emily A. Kirkman (2008-09-20): first version of interactive queries, cleaned up code and generalized many elements to sage.databases.sql_db.py
- Emily A. Kirkman (2007-07-23): inherits GenericSQLDatabase, also added classes: GraphQuery and GenericGraphQuery
- Emily A. Kirkman (2007-05-11): initial sqlite version
- Emily A. Kirkman (2007-02-13): initial version (non-sqlite)

REFERENCES:

- Data provided by Jason Grout (Brigham Young University). [Online] Available: http://artsci.drake.edu/grout/graphs/

class sage.graphs.graph_database.**GenericGraphQuery**(*query_string*, *database=None*, *param_tuple=None*)

Bases: sage.databases.sql_db.SQLQuery

A query for a *GraphDatabase*.

INPUT:

- query_string – a string representing the SQL query
- database – (default: None); the *GraphDatabase* instance to query (if None then a new instance is created)
- param_tuple – a tuple of strings (default: None); what to replace question marks in query_string with (optional, but a good idea)

Note: This query class is generally intended for developers and more advanced users. It allows you to execute any query, and so may be considered unsafe.

EXAMPLES:

See *GraphDatabase* class docstrings or enter:

```
sage: G = GraphDatabase()
sage: G.get_skeleton()
{...
```

to see the underlying structure of the database. Also see sage.databases.sql_db.SQLQuery in sage.databases.sql_db for more info and a tutorial.

A piece of advice about '?' and param_tuple: it is generally considered safer to query with a '?' in place of each value parameter, and using a second argument (a tuple of strings) in a call to the sqlite database. Successful use of the param_tuple argument is exemplified:

```
sage: G = GraphDatabase()
sage: q = 'select graph_id,graph6,num_vertices,num_edges from graph_data where␣
↪graph_id<=(?) and num_vertices=(?)'
sage: param = (22,5)
sage: Q = SQLQuery(G, q, param)
sage: Q.show()
graph_id              graph6                num_vertices          num_edges
------------------------------------------------------------------------------
18                    D??                   5                     0
19                    D?C                   5                     1
20                    D?K                   5                     2
21                    D@O                   5                     2
22                    D?[                   5                     3
```

class sage.graphs.graph_database.**GraphDatabase**

Bases: sage.databases.sql_db.SQLDatabase

Graph Database

This class interfaces with the `sqlite` database `graphs.db`. It is an immutable database that inherits from `SQLDatabase` (see `sage.databases.sql_db`). The display functions and get_graphs_list create their own queries, but it is also possible to query the database by constructing either a `SQLQuery`.

The database contains all unlabeled graphs with 7 or fewer nodes. This class will also interface with the optional database package containing all unlabeled graphs with 8 or fewer nodes. The database consists of five tables. For a full table and column structure, call *graph_db_info()*.

The tables are associated by the unique primary key `graph_id` (int).

To query this database, we create a *GraphQuery*. This can be done directly with the *query()* method or by initializing one of:

- *GenericGraphQuery* – allows direct entry of a query string and tuple of parameters. This is the route for more advanced users that are familiar with SQL
- *GraphQuery* – a wrapper of SQLQuery, a general database/query wrapper of SQLite for new users

REFERENCES:

- Data provided by Jason Grout (Brigham Young University). [Online] Available: http://artsci.drake.edu/grout/graphs/

EXAMPLES:

```
sage: G = GraphDatabase()
sage: G.get_skeleton()
{u'aut_grp': {u'aut_grp_size': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'edge_transitive': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False},
  u'graph_id': {'index': False,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'num_fixed_points': {'index': True,
   'primary_key': False,
```

(continues on next page)

(continued from previous page)

```
   'sql': u'INTEGER',
   'unique': False},
  u'num_orbits': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'vertex_transitive': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False}},
 u'degrees': {u'average_degree': {'index': True,
   'primary_key': False,
   'sql': u'REAL',
   'unique': False},
  u'degree_sequence': {'index': False,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'degrees_sd': {'index': True,
   'primary_key': False,
   'sql': u'REAL',
   'unique': False},
  u'graph_id': {'index': False,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'max_degree': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'min_degree': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'regular': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False}},
 u'graph_data': {u'complement_graph6': {'index': True,
   'primary_key': False,
   'sql': u'TEXT',
   'unique': False},
  u'eulerian': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False},
  u'graph6': {'index': True,
   'primary_key': False,
   'sql': u'TEXT',
   'unique': False},
  u'graph_id': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': True},
  u'lovasz_number': {'index': True,
   'primary_key': False,
   'sql': u'REAL',
```

(continues on next page)

(continued from previous page)

```
   'unique': False},
  u'num_cycles': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'num_edges': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'num_hamiltonian_cycles': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'num_vertices': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'perfect': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False},
  u'planar': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False}},
 u'misc': {u'clique_number': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'diameter': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'edge_connectivity': {'index': True,
   'primary_key': False,
   'sql': u'BOOLEAN',
   'unique': False},
  u'girth': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'graph_id': {'index': False,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'independence_number': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
  u'induced_subgraphs': {'index': True,
   'primary_key': False,
   'sql': u'TEXT',
   'unique': False},
  u'min_vertex_cover_size': {'index': True,
   'primary_key': False,
   'sql': u'INTEGER',
   'unique': False},
```

(continues on next page)

(continued from previous page)

```
 u'num_components': {'index': True,
  'primary_key': False,
  'sql': u'INTEGER',
  'unique': False},
 u'num_cut_vertices': {'index': True,
  'primary_key': False,
  'sql': u'INTEGER',
  'unique': False},
 u'num_spanning_trees': {'index': True,
  'primary_key': False,
  'sql': u'INTEGER',
  'unique': False},
 u'radius': {'index': True,
  'primary_key': False,
  'sql': u'INTEGER',
  'unique': False},
 u'vertex_connectivity': {'index': True,
  'primary_key': False,
  'sql': u'BOOLEAN',
  'unique': False}},
u'spectrum': {u'eigenvalues_sd': {'index': True,
  'primary_key': False,
  'sql': u'REAL',
  'unique': False},
 u'energy': {'index': True,
  'primary_key': False,
  'sql': u'REAL',
  'unique': False},
 u'graph_id': {'index': False,
  'primary_key': False,
  'sql': u'INTEGER',
  'unique': False},
 u'max_eigenvalue': {'index': True,
  'primary_key': False,
  'sql': u'REAL',
  'unique': False},
 u'min_eigenvalue': {'index': True,
  'primary_key': False,
  'sql': u'REAL',
  'unique': False},
 u'spectrum': {'index': False,
  'primary_key': False,
  'sql': u'TEXT',
  'unique': False}}}
```

interactive_query(*display_cols*, ***kwds*)

Generate an interact shell to query the database.

This method generates an interact shell (in the notebook only) that allows the user to manipulate query parameters and see the updated results.

Todo: This function could use improvement. Add full options of typical *GraphQuery* (i.e.: have it accept list input); and update options in interact to make it less annoying to put in operators.

EXAMPLES:

```
sage: D = GraphDatabase()
sage: D.interactive_query(display_cols=['graph6', 'num_vertices', 'degree_
↪sequence'], num_edges=5, max_degree=3)  # py2 # optional -- sagenb
<html>...</html>
```

Warning: Above doctest is known to fail with Python 3 due to `sagenb`. See trac ticket #27435 for more details.

query(*query_dict=None*, *display_cols=None*, ***kwds*)

Create a GraphQuery on this database.

For full class details, type `GraphQuery?` and press `shift+enter`.

EXAMPLES:

```
sage: D = GraphDatabase()
sage: q = D.query(display_cols=['graph6', 'num_vertices', 'degree_sequence'],␣
↪num_edges=['<=', 5])
sage: q.show()
Graph6                Num Vertices          Degree Sequence
-----------------------------------------------------------
@                     1                     [0]
A?                    2                     [0, 0]
A_                    2                     [1, 1]
B?                    3                     [0, 0, 0]
BG                    3                     [0, 1, 1]
BW                    3                     [1, 1, 2]
Bw                    3                     [2, 2, 2]
C?                    4                     [0, 0, 0, 0]
C@                    4                     [0, 0, 1, 1]
CB                    4                     [0, 1, 1, 2]
CF                    4                     [1, 1, 1, 3]
CJ                    4                     [0, 2, 2, 2]
CK                    4                     [1, 1, 1, 1]
CL                    4                     [1, 1, 2, 2]
CN                    4                     [1, 2, 2, 3]
C]                    4                     [2, 2, 2, 2]
C^                    4                     [2, 2, 3, 3]
D??                   5                     [0, 0, 0, 0, 0]
D?C                   5                     [0, 0, 0, 1, 1]
D?K                   5                     [0, 0, 1, 1, 2]
D?[                   5                     [0, 1, 1, 1, 3]
D?{                   5                     [1, 1, 1, 1, 4]
D@K                   5                     [0, 0, 2, 2, 2]
D@O                   5                     [0, 1, 1, 1, 1]
D@S                   5                     [0, 1, 1, 2, 2]
D@[                   5                     [0, 1, 2, 2, 3]
D@s                   5                     [1, 1, 1, 2, 3]
D@{                   5                     [1, 1, 2, 2, 4]
DBW                   5                     [0, 2, 2, 2, 2]
DB[                   5                     [0, 2, 2, 3, 3]
DBg                   5                     [1, 1, 2, 2, 2]
DBk                   5                     [1, 1, 2, 3, 3]
DIk                   5                     [1, 2, 2, 2, 3]
DK[                   5                     [1, 2, 2, 2, 3]
DLo                   5                     [2, 2, 2, 2, 2]
```

(continues on next page)

(continued from previous page)

D_K	5	[1, 1, 1, 1, 2]
D`K	5	[1, 1, 2, 2, 2]
E???	6	[0, 0, 0, 0, 0, 0]
E??G	6	[0, 0, 0, 0, 1, 1]
E??W	6	[0, 0, 0, 1, 1, 2]
E??w	6	[0, 0, 1, 1, 1, 3]
E?@w	6	[0, 1, 1, 1, 1, 4]
E?Bw	6	[1, 1, 1, 1, 1, 5]
E?CW	6	[0, 0, 0, 2, 2, 2]
E?C_	6	[0, 0, 1, 1, 1, 1]
E?Cg	6	[0, 0, 1, 1, 2, 2]
E?Cw	6	[0, 0, 1, 2, 2, 3]
E?Dg	6	[0, 1, 1, 1, 2, 3]
E?Dw	6	[0, 1, 1, 2, 2, 4]
E?Fg	6	[1, 1, 1, 1, 2, 4]
E?Ko	6	[0, 0, 2, 2, 2, 2]
E?Kw	6	[0, 0, 2, 2, 3, 3]
E?LO	6	[0, 1, 1, 2, 2, 2]
E?LW	6	[0, 1, 1, 2, 3, 3]
E?N?	6	[1, 1, 1, 1, 2, 2]
E?NG	6	[1, 1, 1, 1, 3, 3]
E@FG	6	[1, 1, 1, 2, 2, 3]
E@HW	6	[0, 1, 2, 2, 2, 3]
E@N?	6	[1, 1, 2, 2, 2, 2]
E@Ow	6	[0, 1, 2, 2, 2, 3]
E@Q?	6	[1, 1, 1, 1, 1, 1]
E@QW	6	[1, 1, 1, 2, 2, 3]
E@T_	6	[0, 2, 2, 2, 2, 2]
E@YO	6	[1, 1, 2, 2, 2, 2]
EG?W	6	[0, 1, 1, 1, 1, 2]
EGCW	6	[0, 1, 1, 2, 2, 2]
E_?w	6	[1, 1, 1, 1, 1, 3]
E_Cg	6	[1, 1, 1, 1, 2, 2]
E_Cw	6	[1, 1, 1, 2, 2, 3]
E_Ko	6	[1, 1, 2, 2, 2, 2]
F????	7	[0, 0, 0, 0, 0, 0, 0]
F????G	7	[0, 0, 0, 0, 0, 1, 1]
F????W	7	[0, 0, 0, 0, 1, 1, 2]
F????w	7	[0, 0, 0, 1, 1, 1, 3]
F???@w	7	[0, 0, 1, 1, 1, 1, 4]
F???Bw	7	[0, 1, 1, 1, 1, 1, 5]
F???GW	7	[0, 0, 0, 0, 2, 2, 2]
F???G_	7	[0, 0, 0, 1, 1, 1, 1]
F???Gg	7	[0, 0, 0, 1, 1, 2, 2]
F???Gw	7	[0, 0, 0, 1, 2, 2, 3]
F???Hg	7	[0, 0, 1, 1, 1, 2, 3]
F???Hw	7	[0, 0, 1, 1, 2, 2, 4]
F???Jg	7	[0, 1, 1, 1, 1, 2, 4]
F???Wo	7	[0, 0, 0, 2, 2, 2, 2]
F???Ww	7	[0, 0, 0, 2, 2, 3, 3]
F???XO	7	[0, 0, 1, 1, 2, 2, 2]
F???XW	7	[0, 0, 1, 1, 2, 3, 3]
F???Z?	7	[0, 1, 1, 1, 1, 2, 2]
F???ZG	7	[0, 1, 1, 1, 1, 3, 3]
F???^?	7	[1, 1, 1, 1, 1, 2, 3]
F?CJG	7	[0, 1, 1, 1, 2, 2, 3]
F?CPW	7	[0, 0, 1, 2, 2, 2, 3]

(continues on next page)

(continued from previous page)

```
F?CZ?                    7                    [0, 1, 1, 2, 2, 2, 2]
F?C_w                    7                    [0, 0, 1, 2, 2, 2, 3]
F?Ca?                    7                    [0, 1, 1, 1, 1, 1, 1]
F?CaW                    7                    [0, 1, 1, 1, 2, 2, 3]
F?Ch_                    7                    [0, 0, 2, 2, 2, 2, 2]
F?CqO                    7                    [0, 1, 1, 2, 2, 2, 2]
F?LCG                    7                    [1, 1, 1, 1, 2, 2, 2]
F@??W                    7                    [0, 0, 1, 1, 1, 1, 2]
F@?GW                    7                    [0, 0, 1, 1, 2, 2, 2]
FG??w                    7                    [0, 1, 1, 1, 1, 1, 3]
FG?Gg                    7                    [0, 1, 1, 1, 1, 2, 2]
FG?Gw                    7                    [0, 1, 1, 1, 2, 2, 3]
FG?Wo                    7                    [0, 1, 1, 2, 2, 2, 2]
FK??W                    7                    [1, 1, 1, 1, 1, 1, 2]
FK?GW                    7                    [1, 1, 1, 1, 2, 2, 2]
F_?@w                    7                    [1, 1, 1, 1, 1, 1, 4]
F_?Hg                    7                    [1, 1, 1, 1, 1, 2, 3]
F_?XO                    7                    [1, 1, 1, 1, 2, 2, 2]
```

class sage.graphs.graph_database.**GraphQuery**(*graph_db=None*, *query_dict=None*, *display_cols=None*, ***kwds*)

Bases: *sage.graphs.graph_database.GenericGraphQuery*

A query for an instance of *GraphDatabase*.

This class nicely wraps the sage.databases.sql_db.SQLQuery class located in sage.databases.sql_db to make the query constraints intuitive and with as many pre-definitions as possible. (i.e.: since it has to be a *GraphDatabase*, we already know the table structure and types; and since it is immutable, we can treat these as a guarantee).

Note: sage.databases.sql_db.SQLQuery functions are available for *GraphQuery*. See sage.databases.sql_db for more details.

INPUT:

- graph_db – *GraphDatabase* (default: None); instance to apply the query to (If None, then a new instance is created)
- query_dict – dict (default: None); a dictionary specifying the query itself. Format is: {'table_name': 'tblname', 'display_cols': ['col1', 'col2'], 'expression': [col, operator, value]}. If not None, query_dict will take precedence over all other arguments.
- display_cols – list of strings (default: None); a list of column names (strings) to display in the result when running or showing a query
- kwds – the columns of the database are all keywords. For a database table/column structure dictionary, call *graph_db_info()*. Keywords accept both single values and lists of length 2. The list allows the user to specify an expression other than equality. Valid expressions are strings, and for numeric values (i.e. Reals and Integers) are: '=','<','>','>=','<='. String values also accept 'regexp' as an expression argument. The only keyword exception to this format is induced_subgraphs, which accepts one of the following options:
  - ['one_of', String, ..., String] – will search for graphs containing a subgraph isomorphic to *any* of the graph6 strings in the list

– ['all_of', String, ..., String] – will search for graphs containing a subgraph isomorphic to *each* of the graph6 strings in the list

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6', 'num_vertices', 'degree_sequence'],␣
↪num_edges=['<=', 5], min_degree=1)
sage: Q.number_of()
35
sage: Q.show()
Graph6                Num Vertices         Degree Sequence
-------------------------------------------------------------
A_                    2                    [1, 1]
BW                    3                    [1, 1, 2]
CF                    4                    [1, 1, 1, 3]
CK                    4                    [1, 1, 1, 1]
CL                    4                    [1, 1, 2, 2]
CN                    4                    [1, 2, 2, 3]
D?{                   5                    [1, 1, 1, 1, 4]
D@s                   5                    [1, 1, 1, 2, 3]
D@{                   5                    [1, 1, 2, 2, 4]
DBg                   5                    [1, 1, 2, 2, 2]
DBk                   5                    [1, 1, 2, 3, 3]
DIk                   5                    [1, 2, 2, 2, 3]
DK[                   5                    [1, 2, 2, 2, 3]
D_K                   5                    [1, 1, 1, 1, 2]
D`K                   5                    [1, 1, 2, 2, 2]
E?Bw                  6                    [1, 1, 1, 1, 1, 5]
E?Fg                  6                    [1, 1, 1, 1, 2, 4]
E?N?                  6                    [1, 1, 1, 1, 2, 2]
E?NG                  6                    [1, 1, 1, 1, 3, 3]
E@FG                  6                    [1, 1, 1, 2, 2, 3]
E@N?                  6                    [1, 1, 2, 2, 2, 2]
E@Q?                  6                    [1, 1, 1, 1, 1, 1]
E@QW                  6                    [1, 1, 1, 2, 2, 3]
E@YO                  6                    [1, 1, 2, 2, 2, 2]
E_?w                  6                    [1, 1, 1, 1, 1, 3]
E_Cg                  6                    [1, 1, 1, 1, 2, 2]
E_Cw                  6                    [1, 1, 1, 2, 2, 3]
E_Ko                  6                    [1, 1, 2, 2, 2, 2]
F??^?                 7                    [1, 1, 1, 1, 1, 2, 3]
F?LCG                 7                    [1, 1, 1, 1, 2, 2, 2]
FK??W                 7                    [1, 1, 1, 1, 1, 1, 2]
FK?GW                 7                    [1, 1, 1, 1, 2, 2, 2]
F_?@w                 7                    [1, 1, 1, 1, 1, 1, 4]
F_?Hg                 7                    [1, 1, 1, 1, 1, 2, 3]
F_?XO                 7                    [1, 1, 1, 1, 2, 2, 2]
```

get_graphs_list()

Return a list of Sage Graph objects that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6', 'num_vertices', 'degree_sequence
↪'], num_edges=['<=', 5], min_degree=1)
sage: L = Q.get_graphs_list()
sage: L[0]
Graph on 2 vertices
sage: len(L)
```

(continues on next page)

(continued from previous page)

```
35
```

number_of()

Return the number of graphs in the database that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6', 'num_vertices', 'degree_sequence
↪'] ,num_edges=['<=', 5], min_degree=1)
sage: Q.number_of()
35
```

query_iterator()

Return an iterator over the results list of the *GraphQuery*.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6'], num_vertices=7, diameter=5)
sage: for g in Q:
....:     print(g.graph6_string())
F?`po
F?gqg
F@?]O
F@OKg
F@R@o
FA_pW
FEOhW
FGC{o
FIAHo
sage: Q = GraphQuery(display_cols=['graph6'], num_vertices=7, diameter=5)
sage: it = iter(Q)
sage: while True:
....:     try: print(next(it).graph6_string())
....:     except StopIteration: break
F?`po
F?gqg
F@?]O
F@OKg
F@R@o
FA_pW
FEOhW
FGC{o
FIAHo
```

show(*max_field_size=20*, *with_picture=False*)

Display the results of a query in table format.

INPUT:

- `max_field_size` – integer (default: 20); width of fields in command prompt version
- `with_picture` – boolean (default: `False`); whether or not to display results with a picture of the graph (available only in the notebook)

EXAMPLES:

```
sage: G = GraphDatabase()
sage: Q = GraphQuery(G, display_cols=['graph6','num_vertices','aut_grp_size'],
↪ num_vertices=4, aut_grp_size=4)
```

(continues on next page)

(continued from previous page)

```
sage: Q.show()
Graph6                Num Vertices          Aut Grp Size
-------------------------------------------------------------
C@                    4                     4
C^                    4                     4
```

```
sage: R = GraphQuery(G, display_cols=['graph6','num_vertices','degree_sequence
↪'], num_vertices=4)
sage: R.show()
Graph6                Num Vertices          Degree Sequence
-------------------------------------------------------------
C?                    4                     [0, 0, 0, 0]
C@                    4                     [0, 0, 1, 1]
CB                    4                     [0, 1, 1, 2]
CF                    4                     [1, 1, 1, 3]
CJ                    4                     [0, 2, 2, 2]
CK                    4                     [1, 1, 1, 1]
CL                    4                     [1, 1, 2, 2]
CN                    4                     [1, 2, 2, 3]
C]                    4                     [2, 2, 2, 2]
C^                    4                     [2, 2, 3, 3]
C~                    4                     [3, 3, 3, 3]
```

Show the pictures (in notebook mode only):

```
sage: S = GraphQuery(G, display_cols=['graph6','aut_grp_size'], num_
↪vertices=4)
sage: S.show(with_picture=True)
Traceback (most recent call last):
...
NotImplementedError: Cannot display plot on command line.
```

Note that pictures can be turned off:

```
sage: S.show(with_picture=False)
Graph6                Aut Grp Size
-----------------------------------
C?                    24
C@                    4
CB                    2
CF                    6
CJ                    6
CK                    8
CL                    2
CN                    2
C]                    8
C^                    4
C~                    24
```

Show your own query (note that the output is not reformatted for generic queries):

```
sage: (GenericGraphQuery('select degree_sequence from degrees where max_
↪degree=2 and min_degree >= 1', G)).show()
degree_sequence
--------------------
211
```

(continues on next page)

(continued from previous page)

```
222
2211
2222
21111
22211
22211
22222
221111
221111
222211
222211
222211
222222
222222
2111111
2221111
2221111
2221111
2222211
2222211
2222211
2222211
2222222
2222222
```

sage.graphs.graph_database.**data_to_degseq**(*data*, *graph6=None*)

Convert a database integer data type to a degree sequence list.

INPUT:

- `data` – integer data type (one digit per vertex representing its degree, sorted high to low) to be converted to a degree sequence list
- `graph6` – string (default: `None`); the `graph6` identifier is required for all graphs with no edges, so that the correct number of zeros is returned.

EXAMPLES:

```
sage: from sage.graphs.graph_database import data_to_degseq
sage: data_to_degseq(3221)
[1, 2, 2, 3]
sage: data_to_degseq(0, 'D??')
[0, 0, 0, 0, 0]
```

sage.graphs.graph_database.**degseq_to_data**(*degree_sequence*)

Convert a degree sequence list to a sorted (max-min) integer data type.

The input degree sequence list (of Integers) is converted to a sorted (max-min) integer data type, as used for faster access in the underlying database.

INPUT:

- `degree_sequence` – list of integers; input degree sequence list

EXAMPLES:

```
sage: from sage.graphs.graph_database import degseq_to_data
sage: degseq_to_data([2,2,3,1])
3221
```

sage.graphs.graph_database.**graph6_to_plot**(*graph6*)
Return a Graphics object from a graph6 string.

This method constructs a graph from a graph6 string and returns a sage.plot.graphics.Graphics object with arguments preset for the sage.plot.graphics.Graphics.show() method.

INPUT:

- graph6 – a graph6 string

EXAMPLES:

```
sage: from sage.graphs.graph_database import graph6_to_plot
sage: type(graph6_to_plot('D??'))
<class 'sage.plot.graphics.Graphics'>
```

sage.graphs.graph_database.**graph_db_info**(*tablename=None*)
Return a dictionary of allowed table and column names.

INPUT:

- tablename – restricts the output to a single table

EXAMPLES:

```
sage: sorted(graph_db_info())
['aut_grp', 'degrees', 'graph_data', 'misc', 'spectrum']
```

```
sage: graph_db_info(tablename='graph_data')
['complement_graph6',
 'eulerian',
 'graph6',
 'lovasz_number',
 'num_cycles',
 'num_edges',
 'num_hamiltonian_cycles',
 'num_vertices',
 'perfect',
 'planar']
```

sage.graphs.graph_database.**subgraphs_to_query**(*subgraphs*, *db*)
Return a GraphQuery object required for the induced_subgraphs parameter.

This method constructs and returns a *GraphQuery* object respecting the special input required for the induced_subgraphs parameter.

INPUT:

- subgraphs – list of strings; the list should be of one of the following two formats:
  - ['one_of', String, ..., String] – will search for graphs containing a subgraph isomorphic to *any* of the graph6 strings in the list
  - ['all_of', String, ..., String] – will search for graphs containing a subgraph isomorphic to *each* of the graph6 strings in the list
- db – a *GraphDatabase*

Note: This is a helper method called by the *GraphQuery* constructor to handle this special format. This method should not be used on its own because it doesn't set any display columns in the query string, causing a failure to fetch the data when run.

EXAMPLES:

```
sage: from sage.graphs.graph_database import subgraphs_to_query
sage: gd = GraphDatabase()
sage: q = subgraphs_to_query(['all_of', 'A?', 'B?', 'C?'], gd)
sage: q.get_query_string()
'SELECT ,,,,,  FROM misc WHERE ( ( misc.induced_subgraphs regexp ? ) AND (
misc.induced_subgraphs regexp ? ) ) AND ( misc.induced_subgraphs regexp ? )'
```

2.5 Database of strongly regular graphs

This module manages a database associating to a set of four integers (v, k, λ, μ) a strongly regular graphs with these parameters, when one exists.

Using Andries Brouwer's database of strongly regular graphs, it can also return non-existence results. Note that some constructions are missing, and that some strongly regular graphs that exist in the database cannot be automatically built by Sage. Help us if you know any. An outline of the implementation can be found in [**?**].

Note: Any missing/incorrect information in the database must be reported to Andries E. Brouwer directly, in order to have a unique and updated source of information.

REFERENCES:

[BL1984]

2.5.1 Functions

sage.graphs.strongly_regular_db.**SRG_100_44_18_20**()

Return a $(100, 44, 18, 20)$-strongly regular graph.

This graph is built as a Cayley graph, using the construction for Δ_1 with group H_3 presented in Table 8.1 of [JK2003]

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_100_44_18_20
sage: G = SRG_100_44_18_20()                  # long time
sage: G.is_strongly_regular(parameters=True) # long time
(100, 44, 18, 20)
```

sage.graphs.strongly_regular_db.**SRG_100_45_20_20**()

Return a $(100, 45, 20, 20)$-strongly regular graph.

This graph is built as a Cayley graph, using the construction for Γ_3 with group H_3 presented in Table 8.1 of [JK2003].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_100_45_20_20
sage: G = SRG_100_45_20_20()                # long time
sage: G.is_strongly_regular(parameters=True) # long time
(100, 45, 20, 20)
```

sage.graphs.strongly_regular_db.**SRG_105_32_4_12**()
Return a $(105, 32, 4, 12)$-strongly regular graph.

The vertices are the flags of the projective plane of order 4. Two flags (a, A) and (b, B) are adjacent if the point a is on the line B or the point b is on the line A, and $a \neq b$, $A \neq B$. See Theorem 2.7 in [GS1970], and [Coo2006].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_105_32_4_12
sage: G = SRG_105_32_4_12(); G
Aut L(3,4) on flags: Graph on 105 vertices
sage: G.is_strongly_regular(parameters=True)
(105, 32, 4, 12)
```

sage.graphs.strongly_regular_db.**SRG_120_63_30_36**()
Return a $(120, 63, 30, 36)$-strongly regular graph

It is the distance-2 graph of *JohnsonGraph(10,3)*.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_120_63_30_36
sage: G =  SRG_120_63_30_36()
sage: G.is_strongly_regular(parameters=True)
(120, 63, 30, 36)
```

sage.graphs.strongly_regular_db.**SRG_120_77_52_44**()
Return a $(120, 77, 52, 44)$-strongly regular graph.

To build this graph, we first build a $2-(21, 7, 12)$ design, by removing two points from the WittDesign() on 23 points. We then build the intersection graph of blocks with intersection size 3.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_120_77_52_44
sage: G = SRG_120_77_52_44()                  # optional - gap_packages
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages
(120, 77, 52, 44)
```

sage.graphs.strongly_regular_db.**SRG_126_25_8_4**()
Return a $(126, 25, 8, 4)$-strongly regular graph

It is the distance-(1 or 4) graph of *JohnsonGraph(9,4)*.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_126_25_8_4
sage: G =  SRG_126_25_8_4()
sage: G.is_strongly_regular(parameters=True)
(126, 25, 8, 4)
```

sage.graphs.strongly_regular_db.**SRG_126_50_13_24**()
Return a $(126, 50, 13, 24)$-strongly regular graph

This graph is a subgraph of *SRG_175_72_20_36()*. This construction, due to Goethals, is given in §10B.(vii) of [BL1984].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_126_50_13_24
sage: G = SRG_126_50_13_24(); G
Goethals graph: Graph on 126 vertices
sage: G.is_strongly_regular(parameters=True)
(126, 50, 13, 24)
```

sage.graphs.strongly_regular_db.**SRG_1288_792_476_504**()

Return a $(1288, 792, 476, 504)$-strongly regular graph.

This graph is built on the words of weight 12 in the `BinaryGolayCode()`. Two of them are then made adjacent if their symmetric difference has weight 12 (cf [BE1992]).

See also:

strongly_regular_from_two_weight_code() – build a strongly regular graph from a two-weight code.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_1288_792_476_504
sage: G = SRG_1288_792_476_504()              # long time
sage: G.is_strongly_regular(parameters=True) # long time
(1288, 792, 476, 504)
```

sage.graphs.strongly_regular_db.**SRG_144_39_6_12**()

Return a $(144, 39, 6, 12)$-strongly regular graph.

This graph is obtained as an orbit of length 2808 on sets of cardinality 2 (among 2 such orbits) of the group $PGL_3(3)$ acting on the (right) cosets of a subgroup of order 39.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_144_39_6_12
sage: G = SRG_144_39_6_12()
sage: G.is_strongly_regular(parameters=True)
(144, 39, 6, 12)
```

sage.graphs.strongly_regular_db.**SRG_175_72_20_36**()

Return a $(175, 72, 20, 36)$-strongly regular graph

This graph is obtained from the line graph of *HoffmanSingletonGraph()*. Setting two vertices to be adjacent if their distance in the line graph is exactly 2 yields the graph. For more information, see 10.B.(iv) in [BL1984] and https://www.win.tue.nl/~aeb/graphs/McL.html.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_175_72_20_36
sage: G = SRG_175_72_20_36()
sage: G.is_strongly_regular(parameters=True)
(175, 72, 20, 36)
```

sage.graphs.strongly_regular_db.**SRG_176_105_68_54**()

Return a $(176, 105, 68, 54)$-strongly regular graph.

To build this graph, we first build a $2-(22, 7, 16)$ design, by removing one point from the `WittDesign()` on 23 points. We then build the intersection graph of blocks with intersection size 3. Known as S.7 in [Hub1975].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_176_105_68_54
sage: G = SRG_176_105_68_54()                 # optional - gap_packages
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages
(176, 105, 68, 54)
```

sage.graphs.strongly_regular_db.**SRG_176_49_12_14**()

Return a $(176, 49, 12, 14)$-strongly regular graph.

This graph is built from the symmetric Higman-Sims design. In [Bro1982], it is explained that there exists an involution σ exchanging the points and blocks of the Higman-Sims design, such that each point is mapped on a block that contains it (i.e. σ is a 'polarity with all universal points'). The graph is then built by making two vertices u, v adjacent whenever $v \in \sigma(u)$.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_176_49_12_14
sage: G = SRG_176_49_12_14()                  # optional - gap_packages # long time
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages # long time
(176, 49, 12, 14)
```

sage.graphs.strongly_regular_db.**SRG_176_90_38_54**()

Return a $(176, 90, 38, 54)$-strongly regular graph

This graph is obtained from *SRG_175_72_20_36()* by attaching a isolated vertex and doing Seidel switching with respect to disjoint union of 18 maximum cliques, following a construction by W.Haemers given in Sect.10.B.(vi) of [BL1984].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_176_90_38_54
sage: G = SRG_176_90_38_54(); G
a Seidel switching of Distance graph for distance 2 in : Graph on 176 vertices
sage: G.is_strongly_regular(parameters=True)
(176, 90, 38, 54)
```

sage.graphs.strongly_regular_db.**SRG_196_91_42_42**()

Return a $(196, 91, 42, 42)$-strongly regular graph.

This strongly regular graph is built following the construction provided in Corollary 8.2.27 of [IS2006].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_196_91_42_42
sage: G = SRG_196_91_42_42()
sage: G.is_strongly_regular(parameters=True)
(196, 91, 42, 42)
```

sage.graphs.strongly_regular_db.**SRG_210_99_48_45**()

Return a strongly regular graph with parameters $(210, 99, 48, 45)$

This graph is from Example 4.2 in [KPRWZ2010]. One considers the action of the symmetric group S_7 on the 210 digraphs isomorphic to the disjoint union of K_1 and the circulant 6-vertex digraph `digraphs.Circulant(6,[1,4])`. It has 16 orbitals; the package [FK1991] found a megring of them, explicitly described in [KPRWZ2010], resulting in this graph.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_210_99_48_45
sage: g=SRG_210_99_48_45()
```

(continues on next page)

(continued from previous page)

```
sage: g.is_strongly_regular(parameters=True)
(210, 99, 48, 45)
```

sage.graphs.strongly_regular_db.**SRG_220_84_38_28**()
Return a $(220, 84, 38, 28)$-strongly regular graph.

This graph is obtained from the *intersection_graph()* of a BIBD_45_9_8(). This construction appears in VII.11.2 from [**?**]

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_220_84_38_28
sage: g=SRG_220_84_38_28()
sage: g.is_strongly_regular(parameters=True)
(220, 84, 38, 28)
```

sage.graphs.strongly_regular_db.**SRG_243_110_37_60**()
Return a $(243, 110, 37, 60)$-strongly regular graph.

Consider the orthogonal complement of the TernaryGolayCode(), which has 243 words. On them we define a graph, in which two words are adjacent whenever their Hamming distance is 9. This construction appears in [GS1975].

Note: A strongly regular graph with the same parameters is also obtained from the database of 2-weight codes.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_243_110_37_60
sage: G = SRG_243_110_37_60()
sage: G.is_strongly_regular(parameters=True)
(243, 110, 37, 60)
```

sage.graphs.strongly_regular_db.**SRG_253_140_87_65**()
Return a $(253, 140, 87, 65)$-strongly regular graph.

To build this graph, we first build the WittDesign() on 23 points which is a $2-(23, 7, 21)$ design. We then build the intersection graph of blocks with intersection size 3. Known as S.6 in [Hub1975].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_253_140_87_65
sage: G = SRG_253_140_87_65()                # optional - gap_packages
sage: G.is_strongly_regular(parameters=True) # optional - gap_packages
(253, 140, 87, 65)
```

sage.graphs.strongly_regular_db.**SRG_276_140_58_84**()
Return a $(276, 140, 58, 84)$-strongly regular graph.

The graph is built from *McLaughlinGraph()*, with an added isolated vertex. We then perform a *seidel_switching()* on a set of 28 disjoint 5-cliques, which exist by cf. [HT1996].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_276_140_58_84
sage: g=SRG_276_140_58_84()                  # long time # optional - gap_packages
sage: g.is_strongly_regular(parameters=True) # long time # optional - gap_packages
(276, 140, 58, 84)
```

sage.graphs.strongly_regular_db.**SRG_280_117_44_52**()
Return a strongly regular graph with parameters $(280, 117, 44, 52)$.

This graph is built according to a very pretty construction of Mathon and Rosa [MR1985]:

> The vertices of the graph G are all partitions of a set of 9 elements into $\{\{a,b,c\},\{d,e,f\},\{g,h,i\}\}$. The cross-intersection of two such partitions $P = \{P_1, P_2, P_3\}$ and $P' = \{P'_1, P'_2, P'_3\}$ being defined as $\{P_i \cap P'_j : 1 \leq i, j \leq 3\}$, two vertices of G are set to be adjacent if the cross-intersection of their respective partitions does not contain exactly 7 nonempty sets.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_280_117_44_52
sage: g=SRG_280_117_44_52()
sage: g.is_strongly_regular(parameters=True)
(280, 117, 44, 52)
```

sage.graphs.strongly_regular_db.**SRG_280_135_70_60**()
Return a strongly regular graph with parameters $(280, 135, 70, 60)$.

This graph is built from the action of J_2 on the cosets of a $3.PGL(2,9)$-subgroup.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_280_135_70_60
sage: g=SRG_280_135_70_60()                  # long time # optional - internet
sage: g.is_strongly_regular(parameters=True) # long time # optional - internet
(280, 135, 70, 60)
```

sage.graphs.strongly_regular_db.**SRG_416_100_36_20**()
Return a $(416, 100, 36, 20)$-strongly regular graph.

This graph is obtained as an orbit on sets of cardinality 2 (among 2 that exists) of the group $G_2(4)$. This graph is isomorphic to the subgraph of the from *Suzuki Graph* induced on the neighbors of a vertex. Known as S.14 in [Hub1975].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_416_100_36_20
sage: g = SRG_416_100_36_20()                # long time # optional - internet
sage: g.is_strongly_regular(parameters=True) # long time # optional - internet
(416, 100, 36, 20)
```

sage.graphs.strongly_regular_db.**SRG_560_208_72_80**()
Return a $(560, 208, 72, 80)$-strongly regular graph

This graph is obtained as the union of 4 orbits of sets of cardinality 2 (among the 13 that exist) of the group $Sz(8)$.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_560_208_72_80
sage: g = SRG_560_208_72_80()                # not tested (~2s)
sage: g.is_strongly_regular(parameters=True) # not tested (~2s)
(560, 208, 72, 80)
```

sage.graphs.strongly_regular_db.**SRG_630_85_20_10**()
Return a $(630, 85, 20, 10)$-strongly regular graph

This graph is the line graph of $pg(5, 18, 2)$; its point graph is *SRG_175_72_20_36()*. One selects a subset of 630 maximum cliques in the latter following a construction by W.Haemers given in Sect.10.B.(v) of [BL1984].

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import SRG_630_85_20_10
sage: G = SRG_630_85_20_10()                    # long time
sage: G.is_strongly_regular(parameters=True)    # long time
(630, 85, 20, 10)
```

sage.graphs.strongly_regular_db.**SRG_from_RSHCD**(*v*, *k*, *l*, *mu*, *existence=False*, *check=True*)

Return a (v, k, l, mu)-strongly regular graph from a RSHCD

This construction appears in 8.D of [BL1984]. For more information, see `regular_symmetric_hadamard_matrix_with_constant_diagonal()`.

INPUT:

- `v,k,l,mu` (integers)
- `existence` (boolean) – whether to return a graph or to test if Sage can build such a graph.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

some graphs

```
sage: from sage.graphs.strongly_regular_db import SRG_from_RSHCD
sage: SRG_from_RSHCD(784, 0, 14, 38, existence=True)
False
sage: SRG_from_RSHCD(784, 377, 180, 182, existence=True)
True
sage: SRG_from_RSHCD(144, 65, 28, 30)
Graph on 144 vertices
```

an example with vertex-transitive automorphism group, found during the implementation of the case $v = 324$

```
sage: G=SRG_from_RSHCD(324,152,70,72)  # long time
sage: a=G.automorphism_group()         # long time
sage: a.order()                        # long time
2592
sage: len(a.orbits())                  # long time
1
```

sage.graphs.strongly_regular_db.**apparently_feasible_parameters**(*n*)

Return a list of a priori feasible parameters (v, k, λ, μ), with $0 < \mu < k$.

Note that some of those that it returns may also be infeasible for more involved reasons. The condition $0 < \mu < k$ makes sure we skip trivial cases of complete multipartite graphs and their complements.

INPUT:

- `n` (integer) – return all a-priori feasible tuples (v, k, λ, μ) for $v < n$

EXAMPLES:

All sets of parameters with $v < 20$ which pass basic arithmetic tests are feasible:

```
sage: from sage.graphs.strongly_regular_db import apparently_feasible_parameters
sage: small_feasible = apparently_feasible_parameters(20); small_feasible
{(5, 2, 0, 1),
```

(continues on next page)

(continued from previous page)

```
 (9, 4, 1, 2),
 (10, 3, 0, 1),
 (10, 6, 3, 4),
 (13, 6, 2, 3),
 (15, 6, 1, 3),
 (15, 8, 4, 4),
 (16, 5, 0, 2),
 (16, 6, 2, 2),
 (16, 9, 4, 6),
 (16, 10, 6, 6),
 (17, 8, 3, 4)}
sage: all(graphs.strongly_regular_graph(*x,existence=True) is True for x in small_
↪feasible)
True
```

But that becomes wrong for $v < 60$ (because of the non-existence of a $(49, 16, 3, 6)$-strongly regular graph):

```
sage: small_feasible = apparently_feasible_parameters(60)
sage: all(graphs.strongly_regular_graph(*x,existence=True) is True for x in small_
↪feasible)
False
```

sage.graphs.strongly_regular_db.**eigenmatrix**(*v*, *k*, *l*, *mu*)

Return the first eigenmatrix of a (v, k, l, mu)-strongly regular graph.

The adjacency matrix A of an s.r.g. commutes with the adjacency matrix $A' = J - A - I$ of its complement (here J is all-1 matrix, and I the identity matrix). Thus, they can be simultaneously diagonalized and so A and A' share eigenspaces.

The eigenvalues of J are v with multiplicity 1, and 0 with multiplicity $v - 1$. Thus the eigenvalue of A' corresponding to the 1-dimension k-eigenspace of A is $v - k - 1$. Respectively, the eigenvalues of A' corresponding to t-eigenspace of A, with t unequal to k, equals $-t - 1$. The 1st eigenmatrix P of the C-algebra $C[A]$ generated by A encodes this eigenvalue information in its three columns; the 2nd (resp. 3rd) column contains distinct eigenvalues of A (resp. of A'), and the 1st column contains the corresponding eigenvalues of I. The matrix vP^{-1} is called the 2nd eigenvalue matrix of $C[A]$.

The most interesting feature of vP^{-1} is that it is the 1st eigenmatrix of the dual of $C[A]$ if the dual is generated by the adjacency matrix of a strongly regular graph. See [**?**] and [BI1984] for details.

If the set of parameters is not feasible, or if they correspond to a conference graph, the function returns `None`. Its output is stable, assuming that the eigenvalues r,s used satisfy r>s; this holds for the current implementation of eigenvalues().

INPUT:

- `v,k,l,mu` (integers)

EXAMPLES:

Petersen's graph's C-algebra does not have a dual coming from an s.r.g.:

```
sage: from sage.graphs.strongly_regular_db import eigenmatrix
sage: P=eigenmatrix(10,3,0,1); P
[ 1  3  6]
[ 1  1 -2]
[ 1 -2  1]
sage: 10*P^-1
[   1    5    4]
```

(continues on next page)

(continued from previous page)

```
[   1  5/3 -8/3]
[   1 -5/3  2/3]
```

The line graph of $K_{3,3}$ is self-dual:

```
sage: P=eigenmatrix(9,4,1,2); P
[ 1  4  4]
[ 1  1 -2]
[ 1 -2  1]
sage: 9*P^-1
[ 1  4  4]
[ 1  1 -2]
[ 1 -2  1]
```

A strongly regular graph with a non-isomorphic dual coming from another strongly regular graph:

```
sage: graphs.strongly_regular_graph(243,220,199,200, existence=True)
True
sage: graphs.strongly_regular_graph(243,110,37,60, existence=True)
True
sage: P=eigenmatrix(243,220,199,200); P
[  1 220  22]
[  1   4  -5]
[  1  -5   4]
sage: 243*P^-1
[  1 110 132]
[  1   2  -3]
[  1 -25  24]
sage: 243*P^-1==eigenmatrix(243,110,37,60)
True
```

sage.graphs.strongly_regular_db.**is_GQqmqp**(v, k, l, mu)

Test whether some $GQ(q-1,q+1)$ or $GQ(q+1,q-1)$-graph is (v,k,λ,μ)-srg.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_GQqmqp
sage: t = is_GQqmqp(27,10,1,5); t
(<function AhrensSzekeresGeneralizedQuadrangleGraph at ...>, 3, False)
sage: g = t[0](*t[1:]); g
AS(3); GQ(2, 4): Graph on 27 vertices
sage: t = is_GQqmqp(45,12,3,3); t
(<function AhrensSzekeresGeneralizedQuadrangleGraph at ...>, 3, True)
sage: g = t[0](*t[1:]); g
AS(3)*; GQ(4, 2): Graph on 45 vertices
sage: g.is_strongly_regular(parameters=True)
(45, 12, 3, 3)
sage: t = is_GQqmqp(16,6,2,2); t
(<function T2starGeneralizedQuadrangleGraph at ...>, 2, True)
sage: g = t[0](*t[1:]); g
```

(continues on next page)

(continued from previous page)

```
T2*(O,2)*; GQ(3, 1): Graph on 16 vertices
sage: g.is_strongly_regular(parameters=True)
(16, 6, 2, 2)
sage: t = is_GQqmqp(64,18,2,6); t
(<function T2starGeneralizedQuadrangleGraph at ...>, 4, False)
sage: g = t[0](*t[1:]); g
T2*(O,4); GQ(3, 5): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 18, 2, 6)
```

sage.graphs.strongly_regular_db.**is_NO_F2**(*v*, *k*, *l*, *mu*)

Test whether some NO^e,perp(2n,2) graph is (v, k, λ, μ)-strongly regular.

For more information, see *sage.graphs.graph_generators.GraphGenerators.NonisotropicOrthogonalPolarGraph()*.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_NO_F2
sage: t = is_NO_F2(10, 3, 0, 1); t
(<function NonisotropicOrthogonalPolarGraph at ...>, 4, 2, '-')
sage: g = t[0](*t[1:]); g
NO^-(4, 2): Graph on 10 vertices
sage: g.is_strongly_regular(parameters=True)
(10, 3, 0, 1)
```

sage.graphs.strongly_regular_db.**is_NO_F3**(*v*, *k*, *l*, *mu*)

Test whether some NO^e,perp(2n,3) graph is (v, k, λ, μ)-strongly regular.

For more information, see *sage.graphs.graph_generators.GraphGenerators.NonisotropicOrthogonalPolarGraph()*.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_NO_F3
sage: t = is_NO_F3(15, 6, 1, 3); t
(<function NonisotropicOrthogonalPolarGraph at ...>, 4, 3, '-')
sage: g = t[0](*t[1:]); g
NO^-(4, 3): Graph on 15 vertices
sage: g.is_strongly_regular(parameters=True)
(15, 6, 1, 3)
```

sage.graphs.strongly_regular_db.**is_NOodd**(*v*, *k*, *l*, *mu*)

Test whether some NO^e(2n+1,q) graph is (v, k, λ, μ)-strongly regular.

Here $q > 2$, for in the case $q = 2$ this graph is complete. For more information, see *sage.graphs.graph_generators.GraphGenerators.NonisotropicOrthogonalPolarGraph()* and Sect. 7.C of [BL1984].

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_NOodd
sage: t = is_NOodd(120, 51, 18, 24); t
(<function NonisotropicOrthogonalPolarGraph at ...>, 5, 4, '-')
sage: g = t[0](*t[1:]); g
NO^-(5, 4): Graph on 120 vertices
sage: g.is_strongly_regular(parameters=True)
(120, 51, 18, 24)
```

`sage.graphs.strongly_regular_db.`**`is_NOperp_F5`**`(v, k, l, mu)`

Test whether some NO^e,perp(2n+1,5) graph is (v, k, λ, μ)-strongly regular.

For more information, see *sage.graphs.graph_generators.GraphGenerators.NonisotropicOrthogonalPolarGraph()* and Sect. 7.D of [BL1984].

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_NOperp_F5
sage: t = is_NOperp_F5(10, 3, 0, 1); t
(<function NonisotropicOrthogonalPolarGraph at ...>, 3, 5, '-', 1)
sage: g = t[0](*t[1:]); g
NO^-,perp(3, 5): Graph on 10 vertices
sage: g.is_strongly_regular(parameters=True)
(10, 3, 0, 1)
```

`sage.graphs.strongly_regular_db.`**`is_NU`**`(v, k, l, mu)`

Test whether some NU(n,q)-graph, is (v, k, λ, μ)-strongly regular.

Note that n>2; for n=2 there is no s.r.g. For more information, see *sage.graphs.graph_generators.GraphGenerators.NonisotropicUnitaryPolarGraph()* and series C14 in [Hub1975].

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_NU
sage: t = is_NU(40, 27, 18, 18); t
(<function NonisotropicUnitaryPolarGraph at ...>, 4, 2)
sage: g = t[0](*t[1:]); g
NU(4, 2): Graph on 40 vertices
sage: g.is_strongly_regular(parameters=True)
(40, 27, 18, 18)
```

sage.graphs.strongly_regular_db.**is_RSHCD**(*v*, *k*, *l*, *mu*)

Test whether some RSHCD graph is (v, k, λ, μ)-strongly regular.

For more information, see *SRG_from_RSHCD()*.

INPUT:

- v,k,l,mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_RSHCD
sage: t = is_RSHCD(64,27,10,12); t
[<built-in function SRG_from_RSHCD>, 64, 27, 10, 12]
sage: g = t[0](*t[1:]); g
Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 27, 10, 12)
```

sage.graphs.strongly_regular_db.**is_affine_polar**(*v*, *k*, *l*, *mu*)

Test whether some Affine Polar graph is (v, k, λ, μ)-strongly regular.

For more information, see https://www.win.tue.nl/~aeb/graphs/VO.html.

INPUT:

- v,k,l,mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_affine_polar
sage: t = is_affine_polar(81,32,13,12); t
(..., 4, 3)
sage: g = t[0](*t[1:]); g
Affine Polar Graph VO^+(4,3): Graph on 81 vertices
sage: g.is_strongly_regular(parameters=True)
(81, 32, 13, 12)

sage: t = is_affine_polar(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_complete_multipartite**(*v*, *k*, *l*, *mu*)

Test whether some complete multipartite graph is (v, k, λ, μ)-strongly regular.

Any complete multipartite graph with parts of the same size is strongly regular.

INPUT:

- v, k, l, mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_complete_multipartite
sage: t = is_complete_multipartite(12,8,4,8); t
(<cyfunction is_complete_multipartite.<locals>.CompleteMultipartiteSRG at ...>,
 3,
 4)
sage: g = t[0](*t[1:]); g
Multipartite Graph with set sizes [4, 4, 4]: Graph on 12 vertices
sage: g.is_strongly_regular(parameters=True)
(12, 8, 4, 8)
```

sage.graphs.strongly_regular_db.**is_cossidente_penttila**(*v*, *k*, *l*, *mu*)

Test whether some CossidentePenttilaGraph graph is (v, k, λ, μ)-strongly regular.

For more information, see *CossidentePenttilaGraph()*.

INPUT:

- v, k, l, mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_cossidente_penttila
sage: t =  is_cossidente_penttila(378, 52, 1, 8); t
(<function CossidentePenttilaGraph at ...>, 5)
sage: g = t[0](*t[1:]); g                       # optional - gap_packages
CossidentePenttila(5): Graph on 378 vertices
sage: g.is_strongly_regular(parameters=True)    # optional - gap_packages
(378, 52, 1, 8)
```

sage.graphs.strongly_regular_db.**is_goethals_seidel**(*v*, *k*, *l*, *mu*)

Test whether some *GoethalsSeidelGraph()* graph is (v, k, λ, μ)-strongly regular.

INPUT:

- v, k, l, mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_goethals_seidel
sage: t = is_goethals_seidel(28, 15, 6, 10); t
[<function GoethalsSeidelGraph at ...>, 3, 3]
sage: g = t[0](*t[1:]); g
Graph on 28 vertices
sage: g.is_strongly_regular(parameters=True)
(28, 15, 6, 10)

sage: t = is_goethals_seidel(256, 135, 70, 72); t
```

(continues on next page)

(continued from previous page)

```
[<function GoethalsSeidelGraph at ...>, 2, 15]
sage: g = t[0](*t[1:]); g
Graph on 256 vertices
sage: g.is_strongly_regular(parameters=True)
(256, 135, 70, 72)

sage: t = is_goethals_seidel(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_haemers**(*v*, *k*, *l*, *mu*)

Test whether some HaemersGraph graph is (v, k, λ, μ)-strongly regular.

For more information, see *HaemersGraph()*.

INPUT:

- v,k,l,mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_haemers
sage: t = is_haemers(96, 19, 2, 4); t
(<function HaemersGraph at ...>, 4)
sage: g = t[0](*t[1:]); g
Haemers(4): Graph on 96 vertices
sage: g.is_strongly_regular(parameters=True)
(96, 19, 2, 4)
```

sage.graphs.strongly_regular_db.**is_johnson**(*v*, *k*, *l*, *mu*)

Test whether some Johnson graph is (v, k, λ, μ)-strongly regular.

INPUT:

- v,k,l,mu (integers)

OUTPUT:

A tuple t such that t[0](*t[1:]) builds the requested graph if one exists, and None otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_johnson
sage: t = is_johnson(10,6,3,4); t
(..., 5)
sage: g = t[0](*t[1:]); g
Johnson graph with parameters 5,2: Graph on 10 vertices
sage: g.is_strongly_regular(parameters=True)
(10, 6, 3, 4)

sage: t = is_johnson(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_mathon_PC_srg**(*v*, *k*, *l*, *mu*)

Test whether some Mathon's Pseudocyclic s.r.g. is (v, k, λ, μ)-strongly regular.

INPUT:

- v,k,l,mu (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

Todo: The current implementation only gives a subset of all possible graphs that can be obtained using this construction. A full implementation should rely on a database of conference matrices (or, equivalently, on a database of s.r.g.'s with parameters $(4t+1, 2t, t-1, t)$. Currently we make an extra assumption that $4t+1$ is a prime power. The first case where we miss a construction is $t = 11$, where we could (recursively) use the graph for $t = 1$ to construct a graph on 83205 vertices.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_mathon_PC_srg
sage: t = is_mathon_PC_srg(45,22,10,11); t
(..., 1)
sage: g = t[0](*t[1:]); g
Mathon's PC SRG on 45 vertices: Graph on 45 vertices
sage: g.is_strongly_regular(parameters=True)
(45, 22, 10, 11)
```

sage.graphs.strongly_regular_db.**is_muzychuk_S6**(*v*, *k*, *l*, *mu*)

Test whether some Muzychuk S6 graph is (v, k, l, mu)-strongly regular.

Tests whether a *MuzychukS6Graph()* has parameters (v, k, l, mu).

INPUT:

- `v, k, l, mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the required graph if it exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_muzychuk_S6
sage: t = is_muzychuk_S6(378, 116, 34, 36)
sage: G = t[0](*t[1:]); G
Muzychuk S6 graph with parameters (3,3): Graph on 378 vertices
sage: G.is_strongly_regular(parameters=True)
(378, 116, 34, 36)
sage: t = is_muzychuk_S6(5, 5, 5, 5); t
```

sage.graphs.strongly_regular_db.**is_nowhere0_twoweight**(*v*, *k*, *l*, *mu*)

Test whether some graph of nowhere 0 words is (v, k, λ, μ)-strongly regular.

Test whether a *Nowhere0WordsTwoWeightCodeGraph()* is (v, k, λ, μ)-strongly regular.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if the parameters match, and `None` otherwise.

EXAMPLES:

```
sage: graphs.strongly_regular_graph(196, 60, 14, 20)
Nowhere0WordsTwoWeightCodeGraph(8): Graph on 196 vertices
```

sage.graphs.strongly_regular_db.**is_orthogonal_array_block_graph**(*v*, *k*, *l*, *mu*)

Test whether some (pseudo)Orthogonal Array graph is (v, k, λ, μ)-strongly regular.

We know how to construct graphs with parameters of an Orthogonal Array ($OA(m, n)$), also known as Latin squares graphs $L_m(n)$, in several cases where no orthogonal array is known, or even in some cases for which they are known not to exist.

Such graphs are usually called pseudo-Latin squares graphs. Namely, Sage can construct a graph with parameters of an $OA(m, n)$-graph whenever there exists a skew-Hadamard matrix of order $n + 1$, and $m = (n + 1)/2$ or $m = (n - 1)/2$. The construction in the former case is due to Goethals-Seidel [BL1984], and in the latter case due to Pasechnik [Pas1992].

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_orthogonal_array_block_graph
sage: t = is_orthogonal_array_block_graph(64, 35, 18, 20); t
(..., 5, 8)
sage: g = t[0](*t[1:]); g
OA(5,8): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 35, 18, 20)
sage: t=is_orthogonal_array_block_graph(225,98,43,42); t
(..., 4)
sage: g = t[0](*t[1:]); g
Pasechnik Graph_4: Graph on 225 vertices
sage: g.is_strongly_regular(parameters=True)
(225, 98, 43, 42)
sage: t=is_orthogonal_array_block_graph(225,112,55,56); t
(..., 4)
sage: g = t[0](*t[1:]); g
skewhad^2_4: Graph on 225 vertices
sage: g.is_strongly_regular(parameters=True)
(225, 112, 55, 56)

sage: t = is_orthogonal_array_block_graph(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_orthogonal_polar**(*v*, *k*, *l*, *mu*)

Test whether some Orthogonal Polar graph is (v, k, λ, μ)-strongly regular.

For more information, see https://www.win.tue.nl/~aeb/graphs/srghub.html.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_orthogonal_polar
sage: t = is_orthogonal_polar(85, 20, 3, 5); t
(<function OrthogonalPolarGraph at ...>, 5, 4, '')
```

(continues on next page)

(continued from previous page)

```
sage: g = t[0](*t[1:]); g
Orthogonal Polar Graph O(5, 4): Graph on 85 vertices
sage: g.is_strongly_regular(parameters=True)
(85, 20, 3, 5)

sage: t = is_orthogonal_polar(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_paley**(*v*, *k*, *l*, *mu*)

Test whether some Paley graph is (v, k, λ, μ)-strongly regular.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_paley
sage: t = is_paley(13,6,2,3); t
(..., 13)
sage: g = t[0](*t[1:]); g
Paley graph with parameter 13: Graph on 13 vertices
sage: g.is_strongly_regular(parameters=True)
(13, 6, 2, 3)
sage: t = is_paley(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_polhill**(*v*, *k*, *l*, *mu*)

Test whether some graph from [Pol2009] is $(1024, k, \lambda, \mu)$-strongly regular.

Note: This function does not actually explore *all* strongly regular graphs produced in [Pol2009], but only those on 1024 vertices.

John Polhill offered his help if we attempt to write a code to guess, given (v, k, λ, μ), which of his construction must be applied to find the graph.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if the parameters match, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_polhill
sage: t = is_polhill(1024, 231,  38,  56); t
[<cyfunction is_polhill.<locals>.<lambda> at ...>]
sage: g = t[0](*t[1:]); g                    # not tested (too long)
Graph on 1024 vertices
sage: g.is_strongly_regular(parameters=True) # not tested (too long)
(1024, 231, 38, 56)
sage: t = is_polhill(1024, 264,  56,  72); t
[<cyfunction is_polhill.<locals>.<lambda> at ...>]
sage: t = is_polhill(1024, 297,  76,  90); t
```

(continues on next page)

(continued from previous page)

```
[<cyfunction is_polhill.<locals>.<lambda> at ...>]
sage: t = is_polhill(1024, 330,  98, 110); t
[<cyfunction is_polhill.<locals>.<lambda> at ...>]
sage: t = is_polhill(1024, 462, 206, 210); t
[<cyfunction is_polhill.<locals>.<lambda> at ...>]
```

sage.graphs.strongly_regular_db.**is_steiner**(*v*, *k*, *l*, *mu*)

Test whether some Steiner graph is (v, k, λ, μ)-strongly regular.

A Steiner graph is the intersection graph of a Steiner set system. For more information, see https://www.win.tue.nl/~aeb/graphs/S.html.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_steiner
sage: t = is_steiner(26,15,8,9); t
(..., 13, 3)
sage: g = t[0](*t[1:]); g
Intersection Graph: Graph on 26 vertices
sage: g.is_strongly_regular(parameters=True)
(26, 15, 8, 9)

sage: t = is_steiner(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_switch_OA_srg**(*v*, *k*, *l*, *mu*)

Test whether some *switch* $OA(k, n) + *$ is (v, k, λ, μ)-strongly regular.

The "switch* $OA(k, n) + *$ graphs appear on Andries Brouwer's database and are built by adding an isolated vertex to a *`OrthogonalArrayBlockGraph()`*, and a *`Seidel switching`* a set of disjoint n-cocliques.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if the parameters match, and `None` otherwise.

EXAMPLES:

```
sage: graphs.strongly_regular_graph(170, 78, 35, 36) # indirect doctest
Graph on 170 vertices
```

sage.graphs.strongly_regular_db.**is_switch_skewhad**(*v*, *k*, *l*, *mu*)

Test whether some `switch skewhad^2+*` is (v, k, λ, μ)-strongly regular.

The `switch skewhad^2+*` graphs appear on Andries Brouwer's database and are built by adding an isolated vertex to the complement of *`SquaredSkewHadamardMatrixGraph()`*, and a *`Seidel switching`* a set of disjoint n-cocliques.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if the parameters match, and `None` otherwise.

EXAMPLES:

```
sage: graphs.strongly_regular_graph(226, 105, 48, 49)
switch skewhad^2+*_4: Graph on 226 vertices
```

sage.graphs.strongly_regular_db.**is_taylor_twograph_srg**(*v*, *k*, *l*, *mu*)

Test whether some Taylor two-graph SRG is (v, k, λ, μ)-strongly regular.

For more information, see §7E of [BL1984].

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph *TaylorTwographSRG* if the parameters match, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_taylor_twograph_srg
sage: t = is_taylor_twograph_srg(28, 15, 6, 10); t
(<function TaylorTwographSRG at ...>, 3)
sage: g = t[0](*t[1:]); g
Taylor two-graph SRG: Graph on 28 vertices
sage: g.is_strongly_regular(parameters=True)
(28, 15, 6, 10)
sage: t = is_taylor_twograph_srg(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_twograph_descendant_of_srg**(*v*, *k0*, *l*, *mu*)

Test whether some descendant graph of a s.r.g. is (v, k_0, λ, μ)-s.r.g.

We check whether there can exist $(v+1, k, \lambda^*, \mu^*)$-s.r.g. G so that `self` is a descendant graph of the regular two-graph specified by G. Specifically, we must have that $v+1 = 2(2k-\lambda^*-\mu^*)$, and $k_0 = 2(k-\mu^*)$, $\lambda = k+\lambda^*-2\mu^*$, $\mu = k-\mu^*$, which give 2 independent linear conditions, say $k-\mu^* = \mu$ and $\lambda^*-\mu^* = \lambda-\mu$. Further, there is a quadratic relation $2k^2-(v+1+4\mu)k+2v\mu = 0$.

If we can construct such G then we return a function to build a (v, k_0, λ, μ)-s.r.g. For more information, see 10.3 in https://www.win.tue.nl/~aeb/2WF02/spectra.pdf

INPUT:

- `v,k0,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists and is known, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_twograph_descendant_of_srg
sage: t = is_twograph_descendant_of_srg(27, 10, 1, 5); t
(<cyfunction is_twograph_descendant_of_srg.<locals>.la at...
sage: g = t[0](*t[1:]); g  # py2
descendant of complement(Johnson graph with parameters 8,2) at {5, 7}: Graph on␣
↪27 vertices
sage: g = t[0](*t[1:]); g  # py3
```

(continues on next page)

(continued from previous page)

```
descendant of complement(Johnson graph with parameters 8,2) at {0, 1}: Graph on␣
↪27 vertices
sage: g.is_strongly_regular(parameters=True)
(27, 10, 1, 5)
sage: t = is_twograph_descendant_of_srg(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_unitary_dual_polar**(*v*, *k*, *l*, *mu*)

Test whether some Unitary Dual Polar graph is (v, k, λ, μ)-strongly regular.

This must be the U_5(q) on totally isotropic lines. For more information, see https://www.win.tue.nl/~aeb/graphs/srghub.html.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_unitary_dual_polar
sage: t = is_unitary_dual_polar(297, 40, 7, 5); t
(<function UnitaryDualPolarGraph at ...>, 5, 2)
sage: g = t[0](*t[1:]); g
Unitary Dual Polar Graph DU(5, 2); GQ(8, 4): Graph on 297 vertices
sage: g.is_strongly_regular(parameters=True)
(297, 40, 7, 5)
sage: t = is_unitary_dual_polar(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**is_unitary_polar**(*v*, *k*, *l*, *mu*)

Test whether some Unitary Polar graph is (v, k, λ, μ)-strongly regular.

For more information, see https://www.win.tue.nl/~aeb/graphs/srghub.html.

INPUT:

- `v,k,l,mu` (integers)

OUTPUT:

A tuple `t` such that `t[0](*t[1:])` builds the requested graph if one exists, and `None` otherwise.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import is_unitary_polar
sage: t = is_unitary_polar(45, 12, 3, 3); t
(<function UnitaryPolarGraph at ...>, 4, 2)
sage: g = t[0](*t[1:]); g
Unitary Polar Graph U(4, 2); GQ(4, 2): Graph on 45 vertices
sage: g.is_strongly_regular(parameters=True)
(45, 12, 3, 3)

sage: t = is_unitary_polar(5,5,5,5); t
```

sage.graphs.strongly_regular_db.**latin_squares_graph_parameters**(*v*, *k*, *l*, *mu*)

Check whether (v,k,l,mu)-strongly regular graph has parameters of an $L_g(n)$ s.r.g.

Also known as pseudo-OA(n,g) case, i.e. s.r.g. with parameters of an OA(n,g)-graph. Return g and n, if they exist. See Sect. 9.1 of [**?**] for details.

INPUT:

- `v,k,l,mu` – (integers) parameters of the graph

OUTPUT:

- `(g, n)` – parameters of an $L_g(n)$ graph, or $None$

sage.graphs.strongly_regular_db.**strongly_regular_from_two_intersection_set**(M)

Return a strongly regular graph from a 2-intersection set.

A set of points in the projective geometry $PG(k,q)$ is said to be a 2-intersection set if it intersects every hyperplane in either h_1 or h_2 points, where $h_1, h_2 \in$
NN.

From a 2-intersection set S can be defined a strongly-regular graph in the following way:

- Place the points of S on a hyperplane H in $PG(k+1,q)$
- Define the graph G on all points of $PG(k+1,q) \backslash H$
- Make two points of $V(G) = PG(k+1,q) \backslash H$ adjacent if the line going through them intersects S

For more information, see e.g. [CD2013] where this explanation has been taken from.

INPUT:

- M – a $|S| \times k$ matrix with entries in F_q representing the points of the 2-intersection set. We assume that the first non-zero entry of each row is equal to 1, that is, they give points in homogeneous coordinates.

The implementation does not check that S is actually a 2-intersection set.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import strongly_regular_from_two_
↪intersection_set
sage: S = Matrix([(0,0,1),(0,1,0)] + [(1,x^2,x) for x in GF(4,'b')])
sage: g = strongly_regular_from_two_intersection_set(S); g
two-intersection set in PG(3,4): Graph on 64 vertices
sage: g.is_strongly_regular(parameters=True)
(64, 18, 2, 6)
```

sage.graphs.strongly_regular_db.**strongly_regular_from_two_weight_code**(L)

Return a strongly regular graph from a two-weight code.

A code is said to be a *two-weight* code the weight of its nonzero codewords (i.e. their number of nonzero coordinates) can only be one of two integer values w_1, w_2. It is said to be *projective* if the minimum weight of the dual code is ≥ 3. A strongly regular graph can be built from a two-weight projective code with weights w_1, w_2 (assuming $w_1 < w_2$) by adding an edge between any two codewords whose difference has weight w_1. For more information, see [LS1981] or [Del1972].

INPUT:

- `L` – a two-weight linear code, or its generating matrix.

EXAMPLES:

```
sage: from sage.graphs.strongly_regular_db import strongly_regular_from_two_
↪weight_code
sage: x=("100022021001111",
....:    "010011211122000",
....:    "001021112100011",
....:    "000110120222220")
sage: M = Matrix(GF(3),[list(l) for l in x])
```

(continues on next page)

(continued from previous page)

```
sage: G = strongly_regular_from_two_weight_code(LinearCode(M))
sage: G.is_strongly_regular(parameters=True)
(81, 50, 31, 30)
```

sage.graphs.strongly_regular_db.**strongly_regular_graph**(*v*, *k*, *l*, *mu=-1*, *existence=False*, *check=True*)

Return a (v, k, λ, μ)-strongly regular graph.

This function relies partly on Andries Brouwer's database of strongly regular graphs. See the documentation of *sage.graphs.strongly_regular_db* for more information.

INPUT:

- `v,k,l,mu` (integers) – note that `mu`, if unspecified, is automatically determined from `v,k,l`.
- `existence` (boolean;''False'') – instead of building the graph, return:
  - `True` – meaning that a (v, k, λ, μ)-strongly regular graph exists.
  - `Unknown` – meaning that Sage does not know if such a strongly regular graph exists (see `sage.misc.unknown`).
  - `False` – meaning that no such strongly regular graph exists.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

Petersen's graph from its set of parameters:

```
sage: graphs.strongly_regular_graph(10,3,0,1,existence=True)
True
sage: graphs.strongly_regular_graph(10,3,0,1)
complement(Johnson graph with parameters 5,2): Graph on 10 vertices
```

Now without specifying μ:

```
sage: graphs.strongly_regular_graph(10,3,0)
complement(Johnson graph with parameters 5,2): Graph on 10 vertices
```

An obviously infeasible set of parameters:

```
sage: graphs.strongly_regular_graph(5,5,5,5,existence=True)
False
sage: graphs.strongly_regular_graph(5,5,5,5)
Traceback (most recent call last):
...
ValueError: There exists no (5, 5, 5, 5)-strongly regular graph
```

An set of parameters proved in a paper to be infeasible:

```
sage: graphs.strongly_regular_graph(324,57,0,12,existence=True)
False
sage: graphs.strongly_regular_graph(324,57,0,12)
Traceback (most recent call last):
...
EmptySetError: Andries Brouwer's database reports that no (324, 57, 0,
```

(continues on next page)

(continued from previous page)

```
12)-strongly regular graph exists. Comments: <a
href="srgtabrefs.html#GavrilyukMakhnev05">Gavrilyuk & Makhnev</a> ...
```

A set of parameters unknown to be realizable in Andries Brouwer's database:

```
sage: graphs.strongly_regular_graph(324,95,22,30,existence=True)
Unknown
sage: graphs.strongly_regular_graph(324,95,22,30)
Traceback (most recent call last):
...
RuntimeError: Andries Brouwer's database reports that no
(324, 95, 22, 30)-strongly regular graph is known to exist.
Comments:
```

A large unknown set of parameters (not in Andries Brouwer's database):

```
sage: graphs.strongly_regular_graph(1394,175,0,25,existence=True)
Unknown
sage: graphs.strongly_regular_graph(1394,175,0,25)
Traceback (most recent call last):
...
RuntimeError: Sage cannot figure out if a (1394, 175, 0, 25)-strongly
regular graph exists.
```

Test the Claw bound (see 3.D of [BL1984]):

```
sage: graphs.strongly_regular_graph(2058,242,91,20,existence=True)
False
```

sage.graphs.strongly_regular_db.**strongly_regular_graph_lazy**(*v*, *k*, *l*, *mu=-1*, *existence=False*)

return a promise to build an (v, k, l, mu)-srg

Return a promise to build an (v, k, l, mu)-srg as a tuple t, with $t[0]$ a function to evaluate on $*t[1:]$.

Input as in `strongly_regular_graph()`, although without *check*.

2.6 ISGCI: Information System on Graph Classes and their Inclusions

This module implements an interface to the ISGCI database in Sage.

This database gathers information on graph classes and their inclusions in each other. It also contains information on the complexity of several computational problems.

It is available on the GraphClasses.org website maintained by H.N. de Ridder et al.

2.6.1 How to use it?

Presently, it is possible to use this database through the variables and methods present in the *graph_classes* object. For instance:

```
sage: Trees = graph_classes.Tree
sage: Chordal = graph_classes.Chordal
```

Inclusions

It is then possible to check the inclusion of classes inside of others, if the information is available in the database:

```
sage: Trees <= Chordal
True
```

And indeed, trees are chordal graphs.

The ISGCI database is not all-knowing, and so comparing two classes can return `True`, `False`, or `Unknown` (see the documentation of the Unknown truth value).

An *unknown* answer to `A <= B` only means that ISGCI cannot deduce from the information in its database that `A` is a subclass of `B` nor that it is not. For instance, ISGCI does not know at the moment that some chordal graphs are not trees:

```
sage: graph_classes.Chordal <= graph_classes.Tree
Unknown
```

Descriptions

Given a graph class, one can obtain its associated information in the ISGCI database with the *description()* method:

```
sage: Chordal.description()
Class of graphs : Chordal
--------------------------
id                            :  gc_32
name                          :  chordal
type                          :  base

Problems :
-----------
3-Colourability               :  Linear
Clique                        :  Polynomial
Clique cover                  :  Polynomial
Cliquewidth                   :  Unbounded
Cliquewidth expression        :  NP-complete
Colourability                 :  Linear
Cutwidth                      :  NP-complete
Domination                    :  NP-complete
Feedback vertex set           :  Polynomial
Hamiltonian cycle             :  NP-complete
Hamiltonian path              :  NP-complete
Independent set               :  Linear
Maximum bisection             :  Unknown
Maximum cut                   :  NP-complete
Minimum bisection             :  Unknown
Recognition                   :  Linear
Treewidth                     :  Polynomial
Weighted clique               :  Polynomial
Weighted feedback vertex set  :  Unknown
Weighted independent set      :  Linear
```

It is possible to obtain the complete list of the classes stored in ISGCI by calling the *show_all()* method (beware – long output):

```
sage: graph_classes.show_all()
id         | name                                          | type                 |␣
↪smallgraph
-------------------------------------------------------------------------------------------
↪-----------------------------------
gc_309     | $K_4$-minor-free                              | base                 |
gc_541     | $N^*$                                         | base                 |
gc_215     | $N^*$-perfect                                 | base                 |
gc_5       | $P_4$-bipartite                               | base                 |
gc_3       | $P_4$-brittle                                 | base                 |
gc_6       | $P_4$-comparability                           | base                 |
gc_7       | $P_4$-extendible                              | base                 |
...
```

Until a proper search method is implemented, this lets one find classes which do not appear in *graph_classes.**.

To retrieve a class of graph from its ISGCI ID one may use the *get_class()* method:

```
sage: GC = graph_classes.get_class("gc_5")
sage: GC
$P_4$-bipartite graphs
```

Recognition of graphs

The graph classes represented by the ISGCI database can alternatively be used to access recognition algorithms. For instance, in order to check that a given graph is a tree one has the following the options

```
sage: graphs.PathGraph(5) in graph_classes.Tree
True
```

or:

```
sage: graphs.PathGraph(5).is_tree()
True
```

Furthermore, all ISGCI graph classes which are defined by the exclusion of a finite sequence of induced subgraphs benefit from a generic recognition algorithm. For instance

```
sage: g = graphs.PetersenGraph()
sage: g in graph_classes.ClawFree
False
sage: g.line_graph() in graph_classes.ClawFree
True
```

Or directly from ISGCI

```
sage: gc = graph_classes.get_class("gc_441")
sage: gc
diamond-free graphs
sage: graphs.PetersenGraph() in gc
True
```

2.6.2 Predefined classes

graph_classes currently predefines the following graph classes

Class	Related methods
Apex	*is_apex()*, *apex_vertices()*
AT_free	*is_asteroidal_triple_free()*
Biconnected	*is_biconnected()*, blocks_and_cut_vertices(), blocks_and_cuts_tree()
BinaryTrees	*BalancedTree()*, *is_tree()*
Bipartite	*BalancedTree()*, is_bipartite()
Block	*is_block_graph()*, *blocks_and_cut_vertices()*, *RandomBlockGraph()*
Chordal	*is_chordal()*
Claw-Free	*ClawGraph()*
Comparability	
Gallai	*is_gallai_tree()*
Grid	*Grid2dGraph()*, *GridGraph()*
Interval	*RandomIntervalGraph()*, *IntervalGraph()*, *is_interval()*
Line	*line_graph_forbidden_subgraphs()*, *is_line_graph()*
Modular	*modular_decomposition()*
Outerplanar	*is_circular_planar()*
Perfect	*is_perfect()*
Planar	*is_planar()*
Polyhedral	is_polyhedral()
Split	*is_split()*
Tree	*trees()*, *is_tree()*
UnitDisk	*IntervalGraph()*
UnitInterval	*is_interval()*

2.6.3 Sage's view of ISGCI

The database is stored by Sage in two ways.

The classes: the list of all graph classes and their properties is stored in a huge dictionary (see *classes()*). Below is what Sage knows of gc_249:

```
sage: graph_classes.classes()['gc_249']        # random
{'problem':
    {'Independent set': 'Polynomial',
     'Treewidth': 'Unknown',
     'Weighted independent set': 'Polynomial',
     'Cliquewidth expression': 'NP-complete',
     'Weighted clique': 'Polynomial',
     'Clique cover': 'Unknown',
     'Domination': 'NP-complete',
     'Clique': 'Polynomial',
     'Colourability': 'NP-complete',
     'Cliquewidth': 'Unbounded',
     '3-Colourability': 'NP-complete',
     'Recognition': 'Linear'},
 'type': 'base',
 'id': 'gc_249',
 'name': 'line'}
```

The class inclusion digraph: Sage remembers the class inclusions through the inclusion digraph (see *inclusion_digraph()*). Its nodes are ID of ISGCI classes:

```
sage: d = graph_classes.inclusion_digraph()
sage: d.vertices()[-10:]
['gc_990', 'gc_991', 'gc_992', 'gc_993', 'gc_994', 'gc_995', 'gc_996', 'gc_997', 'gc_
↪998', 'gc_999']
```

An arc from `gc1` to `gc2` means that `gc1` is a superclass of `gc2`. This being said, not all edges are stored ! To ensure that a given class is included in another one, we have to check whether there is in the digraph a `path` from the first one to the other:

```
sage: bip_id = graph_classes.Bipartite._gc_id
sage: perfect_id = graph_classes.Perfect._gc_id
sage: d.has_edge(perfect_id, bip_id)
False
sage: d.distance(perfect_id, bip_id)
2
```

Hence bipartite graphs are perfect graphs. We can see how ISGCI obtains this result

```
sage: p = d.shortest_path(perfect_id, bip_id)
sage: len(p) - 1
2
sage: print(p)                    # random
['gc_56', 'gc_76', 'gc_69']
sage: for c in p:
....:     print(graph_classes.get_class(c))
perfect graphs
...
bipartite graphs
```

What ISGCI knows is that perfect graphs contain unimodular graph which contain bipartite graphs. Therefore bipartite graphs are perfect !

Note: The inclusion digraph is **NOT ACYCLIC**. Indeed, several entries exist in the ISGCI database which represent the same graph class, for instance Perfect graphs and Berge graphs:

```
sage: graph_classes.inclusion_digraph().is_directed_acyclic()
False
sage: Berge = graph_classes.get_class("gc_274"); Berge
Berge graphs
sage: Perfect = graph_classes.get_class("gc_56"); Perfect
perfect graphs
sage: Berge <= Perfect
True
sage: Perfect <= Berge
True
sage: Perfect == Berge
True
```

2.6.4 Information for developers

- The database is loaded not *so* large, but it is still preferable to only load it on demand. This is achieved through the cached methods *classes()* and *inclusion_digraph()*.

- Upon the first access to the database, the information is extracted from the XML file and stored in the cache of three methods:
  - `sage.graphs.isgci._classes` (dictionary)
  - `sage.graphs.isgci._inclusions` (list of dictionaries)
  - `sage.graphs.isgci._inclusion_digraph` (DiGraph)

  Note that the digraph is only built if necessary (for instance if the user tries to compare two classes).

Todo: Technical things:

- Query the database for non-inclusion results so that comparisons can return `False`, and implement strict inclusions.
- Implement a proper search method for the classes not listed in *graph_classes*

  See also:

  `sage.graphs.isgci.show_all()`.
- Some of the graph classes appearing in *graph_classes* already have a recognition algorithm implemented in Sage. It would be so nice to be able to write `g in Trees`, `g in Perfect`, `g in Chordal`, ... :-)

Long-term stuff:

- Implement simple accessors for all the information in the ISGCI database (as can be done from the website)
- Implement intersection of graph classes
- Write generic recognition algorithms for specific classes (when a graph class is defined by the exclusion of subgraphs, one can write a generic algorithm checking the existence of each of the graphs, and this method already exists in Sage).
- Improve the performance of Sage's graph library by letting it take advantage of the properties of graph classes. For example, *Graph.independent_set()* could use the library to detect that a given graph is, say, a tree or a planar graph, and use a specialized algorithm for finding an independent set.

2.6.5 AUTHORS:

- H.N. de Ridder et al. (ISGCI database)
- Nathann Cohen (Sage implementation)

2.6.6 Methods

class sage.graphs.isgci.**GraphClass**(*name*, *gc_id*, *recognition_function=None*)

Bases: sage.structure.sage_object.SageObject, sage.structure.unique_representation.CachedRepresentation

An instance of this class represents a Graph Class, matching some entry in the ISGCI database.

EXAMPLES:

Testing the inclusion of two classes:

```
sage: Chordal = graph_classes.Chordal
sage: Trees = graph_classes.Tree
sage: Trees <= Chordal
True
sage: Chordal <= Trees
Unknown
```

description()

Prints the information of ISGCI about the current class.

EXAMPLES:

```
sage: graph_classes.Chordal.description()
Class of graphs : Chordal
------------------------
id                              :  gc_32
name                            :  chordal
type                            :  base

Problems :
-----------
3-Colourability                 :  Linear
Clique                          :  Polynomial
Clique cover                    :  Polynomial
Cliquewidth                     :  Unbounded
Cliquewidth expression          :  NP-complete
Colourability                   :  Linear
Cutwidth                        :  NP-complete
Domination                      :  NP-complete
Feedback vertex set             :  Polynomial
Hamiltonian cycle               :  NP-complete
Hamiltonian path                :  NP-complete
Independent set                 :  Linear
Maximum bisection               :  Unknown
Maximum cut                     :  NP-complete
Minimum bisection               :  Unknown
Recognition                     :  Linear
Treewidth                       :  Polynomial
Weighted clique                 :  Polynomial
Weighted feedback vertex set    :  Unknown
Weighted independent set        :  Linear
```

forbidden_subgraphs()

Returns the list of forbidden induced subgraphs defining the class.

If the graph class is not defined by a *finite* list of forbidden induced subgraphs, `None` is returned instead.

EXAMPLES:

```
sage: graph_classes.Perfect.forbidden_subgraphs()
sage: gc = graph_classes.get_class('gc_62')
sage: gc
claw--free graphs
sage: gc.forbidden_subgraphs()
[Graph on 4 vertices]
sage: gc.forbidden_subgraphs()[0].is_isomorphic(graphs.ClawGraph())
True
```

class sage.graphs.isgci.**GraphClasses**

Bases: sage.structure.unique_representation.UniqueRepresentation

classes()

Returns the graph classes, as a dictionary.

Upon the first call, this loads the database from the local XML file. Subsequent calls are cached.

EXAMPLES:

```
sage: t = graph_classes.classes()
sage: type(t)
<... 'dict'>
sage: sorted(t["gc_151"].keys())
['id', 'name', 'problem', 'type']
sage: t["gc_151"]['name']
'cograph'
sage: t["gc_151"]['problem']['Clique']
{'complexity': 'Linear'}
```

get_class(*id*)

Returns the class corresponding to the given id in the ISGCI database.

INPUT:

- id (string) – the desired class' ID

See also:

show_all()

EXAMPLES:

With an existing id:

```
sage: Cographs = graph_classes.get_class("gc_151")
sage: Cographs
cograph graphs
```

With a wrong id:

```
sage: graph_classes.get_class(-1)
Traceback (most recent call last):
...
ValueError: The given class id does not exist in the ISGCI database. Is the␣
↪db too old ? You can update it with graph_classes.update_db().
```

inclusion_digraph()

Returns the class inclusion digraph

Upon the first call, this loads the database from the local XML file. Subsequent calls are cached.

EXAMPLES:

```
sage: g = graph_classes.inclusion_digraph(); g
Digraph on ... vertices
```

inclusions()

Returns the graph class inclusions

OUTPUT:

a list of dictionaries

Upon the first call, this loads the database from the local XML file. Subsequent calls are cached.

EXAMPLES:

```
sage: t = graph_classes.inclusions()
sage: type(t)
<... 'list'>
sage: t[0]
{'sub': 'gc_1', 'super': 'gc_2'}
```

show_all()

Prints all graph classes stored in ISGCI

EXAMPLES:

```
sage: graph_classes.show_all()
id         | name                                    | type                 |␣
↪smallgraph
-------------------------------------------------------------------------------
↪---------------------------------------
gc_309     | $K_4$--minor--free                      | base                 |
gc_541     | $N^*$                                   | base                 |
gc_215     | $N^*$--perfect                          | base                 |
gc_5       | $P_4$--bipartite                        | base                 |
gc_3       | $P_4$--brittle                          | base                 |
gc_6       | $P_4$--comparability                    | base                 |
gc_7       | $P_4$--extendible                       | base                 |
...
```

smallgraphs()

Returns a dictionary associating a graph to a graph description string.

Upon the first call, this loads the database from the local XML files. Subsequent calls are cached.

EXAMPLES:

```
sage: t = graph_classes.smallgraphs()
sage: t
{'2C_4': Graph on 8 vertices,
 '2K_2': Graph on 4 vertices,
 '2K_3': Graph on 6 vertices,
 '2K_3 + e': Graph on 6 vertices,
 '2K_4': Graph on 8 vertices,
 '2P_3': Graph on 6 vertices,
...
sage: t['fish']
Graph on 6 vertices
```

update_db()

Updates the ISGCI database by downloading the latest version from internet.

This method downloads the ISGCI database from the website GraphClasses.org. It then extracts the zip file and parses its XML content.

Depending on the credentials of the user running Sage when this command is run, one attempt is made at saving the result in Sage's directory so that all users can benefit from it. If the credentials are not sufficient, the XML file are saved instead in the user's directory (in the SAGE_DB folder).

EXAMPLES:

```
sage: graph_classes.update_db() # Not tested -- requires internet
```

sage.graphs.isgci.**graph_classes = <sage.graphs.isgci.GraphClasses object>**

CHAPTER

THREE

LOW-LEVEL IMPLEMENTATION

3.1 Overview of (di)graph data structures

This module contains no code, and describes Sage's data structures for graphs and digraphs. They can be used directly at Cython/C level, or through the *Graph* and *DiGraph* classes (except one)

3.1.1 Data structures

Four data structures are natively available for (di)graphs in Sage:

- *sparse_graph* (default) – for sparse (di)graphs, with a $\log(n)$ edge test, and easy enumeration of neighbors. It is the most general-purpose data structure, though it can have a high memory cost in practice.
  - Supports: Addition/removal of edges/vertices, multiple edges, edge labels and loops.
- *dense_graph* – for dense (di)graphs, with a $O(1)$ edge test, and slow enumeration of neighbors.
  - Supports: addition/removal of edges/vertices, and loops.
  - Does not support: multiple edges and edge labels.
- *static_sparse_graph* – for sparse (di)graphs and very intensive computations (at C-level). It is faster than *sparse_graph* in practice and *much* lighter in memory.
  - Supports: multiple edges, edge labels and loops
  - Does not support: addition/removal of edges/vertices.
- *static_dense_graph* – for dense (di)graphs and very intensive computations (at C-level). It is mostly a wrapper over bitsets.
  - Supports: addition/removal of edges/vertices, and loops.
  - Does not support: multiple edges and edge labels.

For more information, see the data structures' respective pages.

3.1.2 The backends

The *Graph* and *DiGraph* objects delegate the storage of vertices and edges to other objects: the *graph backends*:

```
sage: Graph()._backend
<sage.graphs.base.sparse_graph.SparseGraphBackend object at ...>
```

A (di)graph backend is a simpler (di)graph class having only the most elementary methods (e.g.: add/remove vertices/edges). Its vertices can be arbitrary hashable objects.

The only backend available in Sage is *CGraphBackend*.

3.1.3 CGraph and CGraphBackend

CGraphBackend is the backend of all native data structures that can be used by *Graph* and *DiGraph*. It is extended by:

- *DenseGraphBackend*
- *SparseGraphBackend*
- *StaticSparseBackend*

While a *CGraphBackend* deals with arbitrary (hashable) vertices, it contains a `._cg` attribute of type *CGraph* which only deals with integer vertices.

The *CGraph* data structures available in Sage are:

- *DenseGraph*
- *SparseGraph*
- *StaticSparseCGraph*

See the *c_graph* module for more information.

3.2 Fast compiled graphs

This is a Cython implementation of the base class for sparse and dense graphs in Sage. It is not intended for use on its own. Specific graph types should extend this base class and implement missing functionalities. Whenever possible, specific methods should also be overridden with implementations that suit the graph type under consideration.

For an overview of graph data structures in sage, see *overview*.

3.2.1 Data structure

The class `CGraph` maintains the following variables:

- `cdef int num_verts`
- `cdef int num_arcs`
- `cdef int *in_degrees`
- `cdef int *out_degrees`
- `cdef bitset_t active_vertices`

The bitset `active_vertices` is a list of all available vertices for use, but only the ones which are set are considered to actually be in the graph. The variables `num_verts` and `num_arcs` are self-explanatory. Note that `num_verts` is the number of bits set in `active_vertices`, not the full length of the bitset. The arrays `in_degrees` and `out_degrees` are of the same length as the bitset.

For more information about active vertices, see the documentation for the method *realloc*.

class sage.graphs.base.c_graph.**CGraph**
Bases: object

Compiled sparse and dense graphs.

add_arc(*u*, *v*)
Add arc (u, v) to the graph.

INPUT:

- u, v – non-negative integers, must be in self

EXAMPLES:

On the *CGraph* level, this always produces an error, as there are no vertices:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.add_arc(0, 1)
Traceback (most recent call last):
...
LookupError: vertex (0) is not a vertex of the graph
```

It works, once there are vertices and add_arc_unsafe() is implemented:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.add_arc(4, 7)
Traceback (most recent call last):
...
LookupError: vertex (7) is not a vertex of the graph
sage: G.has_arc(1, 0)
False
sage: G.has_arc(0, 1)
True

sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc(0,1)
sage: G.add_arc(4,7)
Traceback (most recent call last):
...
LookupError: vertex (7) is not a vertex of the graph
sage: G.has_arc(1,0)
False
sage: G.has_arc(0,1)
True
```

add_vertex(*k=-1*)
Adds vertex k to the graph.

INPUT:

- k – nonnegative integer or −1 (default: −1); if $k = -1$, a new vertex is added and the integer used is returned. That is, for $k = -1$, this function will find the first available vertex that is not in self and add that vertex to this graph.

OUTPUT:

- −1 – indicates that no vertex was added because the current allocation is already full or the vertex is out of range.

- nonnegative integer – this vertex is now guaranteed to be in the graph.

See also:

- `add_vertex_unsafe` – add a vertex to a graph. This method is potentially unsafe. You should instead use *add_vertex()*.
- `add_vertices` – add a bunch of vertices to a graph

EXAMPLES:

Adding vertices to a sparse graph:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(3, extra_vertices=3)
sage: G.add_vertex(3)
3
sage: G.add_arc(2, 5)
Traceback (most recent call last):
...
LookupError: vertex (5) is not a vertex of the graph
sage: G.add_arc(1, 3)
sage: G.has_arc(1, 3)
True
sage: G.has_arc(2, 3)
False
```

Adding vertices to a dense graph:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(3, extra_vertices=3)
sage: G.add_vertex(3)
3
sage: G.add_arc(2,5)
Traceback (most recent call last):
...
LookupError: vertex (5) is not a vertex of the graph
sage: G.add_arc(1, 3)
sage: G.has_arc(1, 3)
True
sage: G.has_arc(2, 3)
False
```

Repeatedly adding a vertex using $k = -1$ will allocate more memory as required:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(3, extra_vertices=0)
sage: G.verts()
[0, 1, 2]
sage: for i in range(10):
....:     _ = G.add_vertex(-1);
...
sage: G.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(3, extra_vertices=0)
sage: G.verts()
```

(continues on next page)

(continued from previous page)

```
[0, 1, 2]
sage: for i in range(12):
....:     _ = G.add_vertex(-1);
...
sage: G.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

add_vertices(*verts*)

Add vertices from the iterable `verts`.

INPUT:

- `verts` – an iterable of vertices; value -1 has a special meaning – for each such value an unused vertex name is found, used to create a new vertex and returned.

OUTPUT:

List of generated labels if there is any -1 in `verts`. `None` otherwise.

See also:

- *add_vertex()* – add a vertex to a graph

EXAMPLES:

Adding vertices for sparse graphs:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=4, extra_vertices=4)
sage: S.verts()
[0, 1, 2, 3]
sage: S.add_vertices([3, -1, 4, 9])
[5]
sage: S.verts()
[0, 1, 2, 3, 4, 5, 9]
sage: S.realloc(20)
sage: S.verts()
[0, 1, 2, 3, 4, 5, 9]
```

Adding vertices for dense graphs:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: D = DenseGraph(nverts=4, extra_vertices=4)
sage: D.verts()
[0, 1, 2, 3]
sage: D.add_vertices([3, -1, 4, 9])
[5]
sage: D.verts()
[0, 1, 2, 3, 4, 5, 9]
sage: D.realloc(20)
sage: D.verts()
[0, 1, 2, 3, 4, 5, 9]
```

all_arcs(*u*, *v*)

Return the labels of all arcs from `u` to `v`.

INPUT:

- `u` – integer; the tail of an arc

- v – integer; the head of an arc

OUTPUT:

- Raise NotImplementedError. This method is not implemented at the *CGraph* level. A child class should provide a suitable implementation.

See also:

- *all_arcs* – all_arcs method for sparse graphs.

EXAMPLES:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.all_arcs(0, 1)
Traceback (most recent call last):
...
NotImplementedError
```

check_vertex(*n*)

Check that n is a vertex of self.

This method is different from *has_vertex()*. The current method raises an error if n is not a vertex of this graph. On the other hand, *has_vertex()* returns a boolean to signify whether or not n is a vertex of this graph.

INPUT:

- n – a nonnegative integer representing a vertex

OUTPUT:

- Raise an error if n is not a vertex of this graph

See also:

- *has_vertex()* – determine whether this graph has a specific vertex

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=10, expected_degree=3, extra_vertices=10)
sage: S.check_vertex(4)
sage: S.check_vertex(12)
Traceback (most recent call last):
...
LookupError: vertex (12) is not a vertex of the graph
sage: S.check_vertex(24)
Traceback (most recent call last):
...
LookupError: vertex (24) is not a vertex of the graph
sage: S.check_vertex(-19)
Traceback (most recent call last):
...
LookupError: vertex (-19) is not a vertex of the graph
```

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: D = DenseGraph(nverts=10, extra_vertices=10)
sage: D.check_vertex(4)
```

(continues on next page)

(continued from previous page)

```
sage: D.check_vertex(12)
Traceback (most recent call last):
...
LookupError: vertex (12) is not a vertex of the graph
sage: D.check_vertex(24)
Traceback (most recent call last):
...
LookupError: vertex (24) is not a vertex of the graph
sage: D.check_vertex(-19)
Traceback (most recent call last):
...
LookupError: vertex (-19) is not a vertex of the graph
```

current_allocation()

Report the number of vertices allocated.

OUTPUT:

- The number of vertices allocated. This number is usually different from the order of a graph. We may have allocated enough memory for a graph to hold $n > 0$ vertices, but the order (actual number of vertices) of the graph could be less than n.

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=4, extra_vertices=4)
sage: S.current_allocation()
8
sage: S.add_vertex(6)
6
sage: S.current_allocation()
8
sage: S.add_vertex(10)
10
sage: S.current_allocation()
16
sage: S.add_vertex(40)
Traceback (most recent call last):
...
RuntimeError: requested vertex is past twice the allocated range: use realloc
sage: S.realloc(50)
sage: S.add_vertex(40)
40
sage: S.current_allocation()
50
sage: S.realloc(30)
-1
sage: S.current_allocation()
50
sage: S.del_vertex(40)
sage: S.realloc(30)
sage: S.current_allocation()
30
```

The actual number of vertices in a graph might be less than the number of vertices allocated for the graph:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
```

(continues on next page)

(continued from previous page)

```
sage: G = DenseGraph(nverts=3, extra_vertices=2)
sage: order = len(G.verts())
sage: order
3
sage: G.current_allocation()
5
sage: order < G.current_allocation()
True
```

del_all_arcs(*u*, *v*)

Delete all arcs from `u` to `v`.

INPUT:

- `u` – integer; the tail of an arc.
- `v` – integer; the head of an arc.

EXAMPLES:

On the *CGraph* level, this always produces an error, as there are no vertices:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.del_all_arcs(0,1)
Traceback (most recent call last):
...
LookupError: vertex (0) is not a vertex of the graph
```

It works, once there are vertices and `del_arc_unsafe()` is implemented:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(0,1,0)
sage: G.add_arc_label(0,1,1)
sage: G.add_arc_label(0,1,2)
sage: G.add_arc_label(0,1,3)
sage: G.del_all_arcs(0,1)
sage: G.has_arc(0,1)
False
sage: G.arc_label(0,1)
0
sage: G.del_all_arcs(0,1)

sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.has_arc(0, 1)
True
sage: G.del_all_arcs(0, 1)
sage: G.has_arc(0, 1)
False
```

del_vertex(*v*)

Delete the vertex `v`, along with all edges incident to it.

If `v` is not in `self`, fails silently.

INPUT:

- v – a nonnegative integer representing a vertex

See also:

- del_vertex_unsafe – delete a vertex from a graph. This method is potentially unsafe. Use *del_vertex()* instead.

EXAMPLES:

Deleting vertices of sparse graphs:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(3)
sage: G.add_arc(0, 1)
sage: G.add_arc(0, 2)
sage: G.add_arc(1, 2)
sage: G.add_arc(2, 0)
sage: G.del_vertex(2)
sage: for i in range(2):
....:     for j in range(2):
....:         if G.has_arc(i, j):
....:             print("{} {}".format(i,j))
0 1
sage: G = SparseGraph(3)
sage: G.add_arc(0, 1)
sage: G.add_arc(0, 2)
sage: G.add_arc(1, 2)
sage: G.add_arc(2, 0)
sage: G.del_vertex(1)
sage: for i in range(3):
....:     for j in range(3):
....:         if G.has_arc(i, j):
....:             print("{} {}".format(i,j))
0 2
2 0
```

Deleting vertices of dense graphs:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(4)
sage: G.add_arc(0, 1); G.add_arc(0, 2)
sage: G.add_arc(3, 1); G.add_arc(3, 2)
sage: G.add_arc(1, 2)
sage: G.verts()
[0, 1, 2, 3]
sage: G.del_vertex(3); G.verts()
[0, 1, 2]
sage: for i in range(3):
....:     for j in range(3):
....:         if G.has_arc(i, j):
....:             print("{} {}".format(i,j))
0 1
0 2
1 2
```

If the vertex to be deleted is not in this graph, then fail silently:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(3)
sage: G.verts()
[0, 1, 2]
sage: G.has_vertex(3)
False
sage: G.del_vertex(3)
sage: G.verts()
[0, 1, 2]
```

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.verts()
[0, 1, 2, 3, 4]
sage: G.has_vertex(6)
False
sage: G.del_vertex(6)
sage: G.verts()
[0, 1, 2, 3, 4]
```

has_arc(*u*, *v*)

Check if the arc `(u, v)` is in this graph.

INPUT:

- `u` – integer; the tail of an arc
- `v` – integer; the head of an arc

OUTPUT:

- Print a `Not Implemented!` message. This method is not implemented at the *CGraph* level. A child class should provide a suitable implementation.

EXAMPLES:

On the *CGraph* this always returns `False`:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.has_arc(0, 1)
False
```

It works once `has_arc_unsafe` is implemented:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.has_arc(1, 0)
False
sage: G.has_arc(0, 1)
True

sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(0,1)
sage: G.has_arc(1,0)
False
sage: G.has_arc(0,1)
True
```

has_vertex(*n*)

Determine whether the vertex `n` is in `self`.

This method is different from *check_vertex()*. The current method returns a boolean to signify whether or not `n` is a vertex of this graph. On the other hand, *check_vertex()* raises an error if `n` is not a vertex of this graph.

INPUT:

- `n` – a nonnegative integer representing a vertex

OUTPUT:

- `True` if `n` is a vertex of this graph; `False` otherwise.

See also:

- *check_vertex()* – raise an error if this graph does not contain a specific vertex.

EXAMPLES:

Upon initialization, a *SparseGraph* or *DenseGraph* has the first `nverts` vertices:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=10, expected_degree=3, extra_vertices=10)
sage: S.has_vertex(6)
True
sage: S.has_vertex(12)
False
sage: S.has_vertex(24)
False
sage: S.has_vertex(-19)
False
```

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: D = DenseGraph(nverts=10, extra_vertices=10)
sage: D.has_vertex(6)
True
sage: D.has_vertex(12)
False
sage: D.has_vertex(24)
False
sage: D.has_vertex(-19)
False
```

in_neighbors(*v*)

Return the list of in-neighbors of the vertex `v`.

INPUT:

- `v` – integer representing a vertex of this graph

OUTPUT:

- Raise `NotImplementedError`. This method is not implemented at the *CGraph* level. A child class should provide a suitable implementation.

Note: Due to the implementation of SparseGraph, this method is much more expensive than out_neighbors_unsafe for SparseGraph's.

EXAMPLES:

On the *CGraph* level, this always produces an error, as there are no vertices:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.in_neighbors(0)
Traceback (most recent call last):
...
LookupError: vertex (0) is not a vertex of the graph
```

It works, once there are vertices and `out_neighbors_unsafe()` is implemented:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.add_arc(3, 1)
sage: G.add_arc(1, 3)
sage: G.in_neighbors(1)
[0, 3]
sage: G.in_neighbors(3)
[1]

sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc(0,1)
sage: G.add_arc(3,1)
sage: G.add_arc(1,3)
sage: G.in_neighbors(1)
[0, 3]
sage: G.in_neighbors(3)
[1]
```

out_neighbors(*u*)

Return the list of out-neighbors of the vertex `u`.

INPUT:

- `u` – integer representing a vertex of this graph

EXAMPLES:

On the *CGraph* level, this always produces an error, as there are no vertices:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.out_neighbors(0)
Traceback (most recent call last):
...
LookupError: vertex (0) is not a vertex of the graph
```

It works, once there are vertices and `out_neighbors_unsafe()` is implemented:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.add_arc(1, 2)
sage: G.add_arc(1, 3)
sage: G.out_neighbors(0)
[1]
```

(continues on next page)

(continued from previous page)

```
sage: G.out_neighbors(1)
[2, 3]

sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc(0,1)
sage: G.add_arc(1,2)
sage: G.add_arc(1,3)
sage: G.out_neighbors(0)
[1]
sage: G.out_neighbors(1)
[2, 3]
```

realloc(*total*)

Reallocate the number of vertices to use, without actually adding any.

INPUT:

- `total` – integer; the total size to make the array of vertices

OUTPUT:

- Raise a `NotImplementedError`. This method is not implemented in this base class. A child class should provide a suitable implementation.

See also:

- *realloc* – a `realloc` implementation for sparse graphs.
- *realloc* – a `realloc` implementation for dense graphs.

EXAMPLES:

First, note that *realloc()* is implemented for *SparseGraph* and *DenseGraph* differently, and is not implemented at the *CGraph* level:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.realloc(20)
Traceback (most recent call last):
...
NotImplementedError
```

The `realloc` implementation for sparse graphs:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=4, extra_vertices=4)
sage: S.current_allocation()
8
sage: S.add_vertex(6)
6
sage: S.current_allocation()
8
sage: S.add_vertex(10)
10
sage: S.current_allocation()
16
sage: S.add_vertex(40)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
RuntimeError: requested vertex is past twice the allocated range: use realloc
sage: S.realloc(50)
sage: S.add_vertex(40)
40
sage: S.current_allocation()
50
sage: S.realloc(30)
-1
sage: S.current_allocation()
50
sage: S.del_vertex(40)
sage: S.realloc(30)
sage: S.current_allocation()
30
```

The `realloc` implementation for dense graphs:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: D = DenseGraph(nverts=4, extra_vertices=4)
sage: D.current_allocation()
8
sage: D.add_vertex(6)
6
sage: D.current_allocation()
8
sage: D.add_vertex(10)
10
sage: D.current_allocation()
16
sage: D.add_vertex(40)
Traceback (most recent call last):
...
RuntimeError: requested vertex is past twice the allocated range: use realloc
sage: D.realloc(50)
sage: D.add_vertex(40)
40
sage: D.current_allocation()
50
sage: D.realloc(30)
-1
sage: D.current_allocation()
50
sage: D.del_vertex(40)
sage: D.realloc(30)
sage: D.current_allocation()
30
```

verts()

Return a list of the vertices in `self`.

OUTPUT:

- A list of all vertices in this graph

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=4, extra_vertices=4)
sage: S.verts()
[0, 1, 2, 3]
sage: S.add_vertices([3,5,7,9])
sage: S.verts()
[0, 1, 2, 3, 5, 7, 9]
sage: S.realloc(20)
sage: S.verts()
[0, 1, 2, 3, 5, 7, 9]
```

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(3, extra_vertices=2)
sage: G.verts()
[0, 1, 2]
sage: G.del_vertex(0)
sage: G.verts()
[1, 2]
```

class sage.graphs.base.c_graph.**CGraphBackend**

Bases: *sage.graphs.base.graph_backends.GenericGraphBackend*

Base class for sparse and dense graph backends.

```
sage: from sage.graphs.base.c_graph import CGraphBackend
```

This class is extended by *SparseGraphBackend* and *DenseGraphBackend*, which are fully functional backends. This class is mainly just for vertex functions, which are the same for both. A *CGraphBackend* will not work on its own:

```
sage: from sage.graphs.base.c_graph import CGraphBackend
sage: CGB = CGraphBackend()
sage: CGB.degree(0, True)
Traceback (most recent call last):
...
TypeError: 'NoneType' object is not iterable
```

The appropriate way to use these backends is via Sage graphs:

```
sage: G = Graph(30)
sage: G.add_edges([(0,1), (0,3), (4,5), (9, 23)])
sage: G.edges(labels=False)
[(0, 1), (0, 3), (4, 5), (9, 23)]
```

This class handles the labels of vertices and edges. For vertices it uses two dictionaries `vertex_labels` and `vertex_ints`. They are just opposite of each other: `vertex_ints` makes a translation from label to integers (that are internally used) and `vertex_labels` make the translation from internally used integers to actual labels. This class tries hard to avoid translation if possible. This will work only if the graph is built on integers from 0 to $n - 1$ and the vertices are basically added in increasing order.

See also:

- *SparseGraphBackend* – backend for sparse graphs.
- *DenseGraphBackend* – backend for dense graphs.

add_vertex(*name*)

Add a vertex to `self`.

INPUT:

- `name` – the vertex to be added (must be hashable). If `None`, a new name is created.

OUTPUT:

- If `name = None`, the new vertex name is returned. `None` otherwise.

See also:

- *add_vertices()* – add a bunch of vertices of this graph
- *has_vertex()* – returns whether or not this graph has a specific vertex

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_vertex(10)
sage: D.add_vertex([])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

```
sage: S = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: S.add_vertex(10)
sage: S.add_vertex([])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

add_vertices(*vertices*)

Add vertices to `self`.

INPUT:

- `vertices` – iterator of vertex labels; a new name is created, used and returned in the output list for all `None` values in `vertices`

OUTPUT:

Generated names of new vertices if there is at least one `None` value present in `vertices`. `None` otherwise.

See also:

- *add_vertex()* – add a vertex to this graph

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(1)
sage: D.add_vertices([1, 2, 3])
sage: D.add_vertices([None] * 4)
[4, 5, 6, 7]
```

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(0)
sage: G.add_vertices([0, 1])
sage: list(G.iterator_verts(None))
[0, 1]
sage: list(G.iterator_edges([0, 1], True))
[]
```

```
sage: import sage.graphs.base.dense_graph
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_vertices([10, 11, 12])
```

bidirectional_dijkstra(*x*, *y*, *weight_function=None*, *distance_flag=False*)

Return the shortest path or distance from `x` to `y` using a bidirectional version of Dijkstra's algorithm.

INPUT:

- `x` – the starting vertex in the shortest path from `x` to `y`
- `y` – the end vertex in the shortest path from `x` to `y`
- `weight_function` – function (default: `None`); a function that inputs an edge `(u, v, l)` and outputs its weight. If `None`, we use the edge label `l` as a weight.
- `distance_flag` – boolean (default: `False`); when set to `True`, the shortest path distance from `x` to `y` is returned instead of the path.

OUTPUT:

- A list of vertices in the shortest path from `x` to `y` or distance from `x` to `y` is returned depending upon the value of parameter `distance_flag`

EXAMPLES:

```
sage: G = Graph(graphs.PetersenGraph())
sage: for (u, v) in G.edges(labels=None):
....:     G.set_edge_label(u, v, 1)
sage: G.shortest_path(0, 1, by_weight=True)
[0, 1]
sage: G.shortest_path_length(0, 1, by_weight=True)
1
sage: G = DiGraph([(1, 2, {'weight':1}), (1, 3, {'weight':5}), (2, 3, {'weight
↪':1})])
sage: G.shortest_path(1, 3, weight_function=lambda e:e[2]['weight'])
[1, 2, 3]
sage: G.shortest_path_length(1, 3, weight_function=lambda e:e[2]['weight'])
2
```

bidirectional_dijkstra_special(*x*, *y*, *weight_function=None*, *exclude_vertices=None*, *exclude_edges=None*, *include_vertices=None*, *distance_flag=False*, *reduced_weight=None*)

Return the shortest path or distance from `x` to `y` using a bidirectional version of Dijkstra's algorithm.

This method is an extension of *bidirectional_dijkstra()* method enabling to exclude vertices and/or edges from the search for the shortest path between `x` and `y`.

This method also has `include_vertices` option enabling to include the vertices which will be used to search for the shortest path between `x` and `y`.

INPUT:

- `x` – the starting vertex in the shortest path from `x` to `y`
- `y` – the end vertex in the shortest path from `x` to `y`
- `exclude_vertices` – iterable conatiner (default: `None`); iterable of vertices to exclude from the graph while calculating the shortest path from `x` to `y`
- `exclude_edges` – iterable conatiner (default: `None`); iterable of edges to exclude from the graph while calculating the shortest path from `x` to `y`

- include_vertices – iterable conatiner (default: None); iterable of vertices to consider in the graph while calculating the shortest path from x to y
- weight_function – function (default: None); a function that inputs an edge (u, v, l) and outputs its weight. If None, we use the edge label l as a weight.
- distance_flag – boolean (default: False); when set to True, the shortest path distance from x to y is returned instead of the path.
- reduced_weight – dictionary (default: None); a dictionary that takes as input an edge (u, v) and outputs its reduced weight.

OUTPUT:

- A list of vertices in the shortest path from x to y or distance from x to y is returned depending upon the value of parameter distance_flag

EXAMPLES:

```
sage: G = Graph([(1, 2, 20), (2, 3, 10), (3, 4, 30), (1, 5, 20), (5, 6, 10),␣
↪(6, 4, 50), (4, 7, 5)])
sage: G._backend.bidirectional_dijkstra_special(1, 4, weight_function=lambda␣
↪e:e[2])
[1, 2, 3, 4]
sage: G._backend.bidirectional_dijkstra_special(1, 4, weight_function=lambda␣
↪e:e[2], exclude_vertices=[2], exclude_edges=[(3, 4)])
[1, 5, 6, 4]
sage: G._backend.bidirectional_dijkstra_special(1, 4, weight_function=lambda␣
↪e:e[2], exclude_vertices=[2, 7])
[1, 5, 6, 4]
sage: G._backend.bidirectional_dijkstra_special(1, 4, weight_function=lambda␣
↪e:e[2],  exclude_edges=[(5, 6)])
[1, 2, 3, 4]
sage: G._backend.bidirectional_dijkstra_special(1, 4, weight_function=lambda␣
↪e:e[2],  include_vertices=[1, 5, 6, 4])
[1, 5, 6, 4]
```

breadth_first_search(*v*, *reverse=False*, *ignore_direction=False*)

Return a breadth-first search from vertex v.

INPUT:

- v – a vertex from which to start the breadth-first search
- reverse – boolean (default: False); this is only relevant to digraphs. If this is a digraph, consider the reversed graph in which the out-neighbors become the in-neighbors and vice versa.
- ignore_direction – boolean (default: False); this is only relevant to digraphs. If this is a digraph, ignore all orientations and consider the graph as undirected.

ALGORITHM:

Below is a general template for breadth-first search.

- **Input:** A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.
- **Output:** A list D of distances of all vertices from s. A tree T rooted at s.

1. $Q \leftarrow [s]$ # a queue of nodes to visit
2. $D \leftarrow [\infty, \infty, \ldots, \infty]$ # n copies of ∞
3. $D[s] \leftarrow 0$

4. $T \leftarrow [\,]$
5. while length$(Q) > 0$ do
    1. $v \leftarrow$ dequeue(Q)
    2. for each $w \in$ adj(v) do # for digraphs, use out-neighbor set oadj(v)
        1. if $D[w] = \infty$ then
            1. $D[w] \leftarrow D[v] + 1$
            2. enqueue(Q, w)
            3. append(T, vw)
6. return (D, T)

See also:

- *breadth_first_search* – breadth-first search for generic graphs.
- *depth_first_search* – depth-first search for generic graphs.
- *depth_first_search()* – depth-first search for fast compiled graphs.

EXAMPLES:

Breadth-first search of the Petersen graph starting at vertex 0:

```
sage: G = Graph(graphs.PetersenGraph())
sage: list(G.breadth_first_search(0))
[0, 1, 4, 5, 2, 6, 3, 9, 7, 8]
```

Visiting European countries using breadth-first search:

```
sage: G = graphs.EuropeMap(continental=True)
sage: list(G.breadth_first_search("Portugal"))
['Portugal', 'Spain', ..., 'Greece']
```

c_graph()

Return the `._cg` and `._cg_rev` attributes

EXAMPLES:

```
sage: cg,cg_rev = graphs.PetersenGraph()._backend.c_graph()
sage: cg
<sage.graphs.base.sparse_graph.SparseGraph object at ...>
sage: cg_rev
<sage.graphs.base.sparse_graph.SparseGraph object at ...>
```

degree(*v*, *directed*)

Return the degree of the vertex `v`.

INPUT:

- `v` – a vertex of the graph
- `directed` – boolean; whether to take into account the orientation of this graph in counting the degree of `v`

OUTPUT:

- The degree of vertex `v`

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraphBackend
sage: B = SparseGraphBackend(7)
sage: B.degree(3, False)
0
```

del_vertex(*v*)

Delete a vertex in `self`, failing silently if the vertex is not in the graph.

INPUT:

- `v` – vertex to be deleted

See also:

- *del_vertices()* – delete a bunch of vertices from this graph

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.del_vertex(0)
sage: D.has_vertex(0)
False
```

```
sage: S = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: S.del_vertex(0)
sage: S.has_vertex(0)
False
```

del_vertices(*vertices*)

Delete vertices from an iterable container.

INPUT:

- `vertices` – iterator of vertex labels

OUTPUT:

- Same as for *del_vertex()*.

See also:

- *del_vertex()* – delete a vertex of this graph

EXAMPLES:

```
sage: import sage.graphs.base.dense_graph
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.del_vertices([7, 8])
sage: D.has_vertex(7)
False
sage: D.has_vertex(6)
True
```

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.del_vertices([1, 2, 3])
sage: D.has_vertex(1)
False
```

(continues on next page)

(continued from previous page)

```
sage: D.has_vertex(0)
True
```

depth_first_search(*v*, *reverse=False*, *ignore_direction=False*)

Return a depth-first search from vertex `v`.

INPUT:

- `v` – a vertex from which to start the depth-first search
- `reverse` – boolean (default: `False`); this is only relevant to digraphs. If this is a digraph, consider the reversed graph in which the out-neighbors become the in-neighbors and vice versa.
- `ignore_direction` – boolean (default: `False`); this is only relevant to digraphs. If this is a digraph, ignore all orientations and consider the graph as undirected.

ALGORITHM:

Below is a general template for depth-first search.

- **Input:** A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.
- **Output:** A list D of distances of all vertices from s. A tree T rooted at s.

1. $S \leftarrow [s]$ # a stack of nodes to visit
2. $D \leftarrow [\infty, \infty, \ldots, \infty]$ # n copies of ∞
3. $D[s] \leftarrow 0$
4. $T \leftarrow [\,]$
5. while length$(S) > 0$ do
    1. $v \leftarrow$ pop(S)
    2. for each $w \in$ adj(v) do # for digraphs, use out-neighbor set oadj(v)
        1. if $D[w] = \infty$ then
            1. $D[w] \leftarrow D[v] + 1$
            2. push(S, w)
            3. append(T, vw)
6. return (D, T)

See also:

- *breadth_first_search()* – breadth-first search for fast compiled graphs.
- *breadth_first_search* – breadth-first search for generic graphs.
- *depth_first_search* – depth-first search for generic graphs.

EXAMPLES:

Traversing the Petersen graph using depth-first search:

```
sage: G = Graph(graphs.PetersenGraph())
sage: list(G.depth_first_search(0))
[0, 5, 8, 6, 9, 7, 2, 3, 4, 1]
```

Visiting German cities using depth-first search:

```
sage: G = Graph({"Mannheim": ["Frankfurt","Karlsruhe"],
....: "Frankfurt": ["Mannheim","Wurzburg","Kassel"],
....: "Kassel": ["Frankfurt","Munchen"],
....: "Munchen": ["Kassel","Nurnberg","Augsburg"],
....: "Augsburg": ["Munchen","Karlsruhe"],
....: "Karlsruhe": ["Mannheim","Augsburg"],
....: "Wurzburg": ["Frankfurt","Erfurt","Nurnberg"],
....: "Nurnberg": ["Wurzburg","Stuttgart","Munchen"],
....: "Stuttgart": ["Nurnberg"], "Erfurt": ["Wurzburg"]})
sage: list(G.depth_first_search("Stuttgart"))  # py2
['Stuttgart', 'Nurnberg', 'Wurzburg', 'Frankfurt', 'Kassel', 'Munchen',
↪'Augsburg', 'Karlsruhe', 'Mannheim', 'Erfurt']
sage: list(G.depth_first_search("Stuttgart"))  # py3
['Stuttgart', 'Nurnberg', ...]
```

has_vertex(*v*)

Check whether `v` is a vertex of `self`.

INPUT:

- `v` – any object

OUTPUT:

- `True` if `v` is a vertex of this graph; `False` otherwise

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraphBackend
sage: B = SparseGraphBackend(7)
sage: B.has_vertex(6)
True
sage: B.has_vertex(7)
False
```

in_degree(*v*)

Return the in-degree of `v`

INPUT:

- `v` – a vertex of the graph

EXAMPLES:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] }␣
↪)
sage: D.out_degree(1)
2
```

is_connected()

Check whether the graph is connected.

EXAMPLES:

Petersen's graph is connected:

```
sage: DiGraph(graphs.PetersenGraph()).is_connected()
True
```

While the disjoint union of two of them is not:

```
sage: DiGraph(2*graphs.PetersenGraph()).is_connected()
False
```

A graph with non-integer vertex labels:

```
sage: Graph(graphs.CubeGraph(3)).is_connected()
True
```

is_directed_acyclic(*certificate=False*)

Check whether the graph is both directed and acyclic (possibly with a certificate)

INPUT:

- `certificate` – boolean (default: `False`); whether to return a certificate

OUTPUT:

When `certificate=False`, returns a boolean value. When `certificate=True` :

- If the graph is acyclic, returns a pair `(True, ordering)` where `ordering` is a list of the vertices such that `u` appears before `v` in `ordering` if `u, v` is an edge.
- Else, returns a pair `(False, cycle)` where `cycle` is a list of vertices representing a circuit in the graph.

ALGORITHM:

We pick a vertex at random, think hard and find out that that if we are to remove the vertex from the graph we must remove all of its out-neighbors in the first place. So we put all of its out-neighbours in a stack, and repeat the same procedure with the vertex on top of the stack (when a vertex on top of the stack has no out-neighbors, we remove it immediately). Of course, for each vertex we only add its outneighbors to the end of the stack once : if for some reason the previous algorithm leads us to do it twice, it means we have found a circuit.

We keep track of the vertices whose out-neighborhood has been added to the stack once with a variable named `tried`.

There is no reason why the graph should be empty at the end of this procedure, so we run it again on the remaining vertices until none are left or a circuit is found.

Note: The graph is assumed to be directed. An exception is raised if it is not.

EXAMPLES:

At first, the following graph is acyclic:

```
sage: D = DiGraph({ 0:[1,2,3], 4:[2,5], 1:[8], 2:[7], 3:[7], 5:[6,7], 7:[8],␣
↪6:[9], 8:[10], 9:[10] })
sage: D.plot(layout='circular').show()
sage: D.is_directed_acyclic()
True
```

Adding an edge from 9 to 7 does not change it:

```
sage: D.add_edge(9,7)
sage: D.is_directed_acyclic()
True
```

We can obtain as a proof an ordering of the vertices such that u appears before v if uv is an edge of the graph:

```
sage: D.is_directed_acyclic(certificate = True)
(True, [4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10])
```

Adding an edge from 7 to 4, though, makes a difference:

```
sage: D.add_edge(7,4)
sage: D.is_directed_acyclic()
False
```

Indeed, it creates a circuit $7, 4, 5$:

```
sage: D.is_directed_acyclic(certificate = True)
(False, [7, 4, 5])
```

Checking acyclic graphs are indeed acyclic

```
sage: def random_acyclic(n, p):
....:  g = graphs.RandomGNP(n, p)
....:  h = DiGraph()
....:  h.add_edges([ ((u,v) if u<v else (v,u)) for u,v,_ in g.edges() ])
....:  return h
...
sage: all( random_acyclic(100, .2).is_directed_acyclic()     # long time
....:      for i in range(50))                                # long time
True
```

is_strongly_connected()

Check whether the graph is strongly connected.

EXAMPLES:

The circuit on 3 vertices is obviously strongly connected:

```
sage: g = DiGraph({0: [1], 1: [2], 2: [0]})
sage: g.is_strongly_connected()
True
```

But a transitive triangle is not:

```
sage: g = DiGraph({0: [1,2], 1: [2]})
sage: g.is_strongly_connected()
False
```

iterator_in_nbrs(*v*)

Return an iterator over the incoming neighbors of `v`.

INPUT:

- `v` – a vertex of this graph

OUTPUT:

- An iterator over the in-neighbors of the vertex `v`

See also:

- *iterator_nbrs()* – returns an iterator over the neighbors of a vertex
- *iterator_out_nbrs()* – returns an iterator over the out-neighbors of a vertex

EXAMPLES:

```
sage: P = DiGraph(graphs.PetersenGraph().to_directed())
sage: list(P._backend.iterator_in_nbrs(0))
[1, 4, 5]
```

iterator_nbrs(*v*)

Return an iterator over the neighbors of `v`.

INPUT:

- `v` – a vertex of this graph

OUTPUT:

- An iterator over the neighbors the vertex `v`

See also:

- *iterator_in_nbrs()* – returns an iterator over the in-neighbors of a vertex
- *iterator_out_nbrs()* – returns an iterator over the out-neighbors of a vertex
- *iterator_verts()* – returns an iterator over a given set of vertices

EXAMPLES:

```
sage: P = Graph(graphs.PetersenGraph())
sage: list(P._backend.iterator_nbrs(0))
[1, 4, 5]
```

iterator_out_nbrs(*v*)

Return an iterator over the outgoing neighbors of `v`.

INPUT:

- `v` – a vertex of this graph

OUTPUT:

- An iterator over the out-neighbors of the vertex `v`

See also:

- *iterator_nbrs()* – returns an iterator over the neighbors of a vertex
- *iterator_in_nbrs()* – returns an iterator over the in-neighbors of a vertex

EXAMPLES:

```
sage: P = DiGraph(graphs.PetersenGraph().to_directed())
sage: list(P._backend.iterator_out_nbrs(0))
[1, 4, 5]
```

iterator_verts(*verts=None*)

Return an iterator over the vertices of `self` intersected with `verts`.

INPUT:

- `verts` – an iterable container of objects (default: `None`)

OUTPUT:

- If `verts=None`, return an iterator over all vertices of this graph

- If `verts` is a single vertex of the graph, treat it as the container `[verts]`
- If `verts` is a iterable container of vertices, find the intersection of `verts` with the vertex set of this graph and return an iterator over the resulting intersection

See also:

- *`iterator_nbrs()`* – returns an iterator over the neighbors of a vertex.

EXAMPLES:

```
sage: P = Graph(graphs.PetersenGraph())
sage: list(P._backend.iterator_verts(P))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list(P._backend.iterator_verts())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list(P._backend.iterator_verts([1, 2, 3]))
[1, 2, 3]
sage: list(P._backend.iterator_verts([1, 2, 10]))
[1, 2]
```

loops (*new=None*)

Check whether loops are allowed in this graph.

INPUT:

- `new` – boolean (default: `None`); to set or `None` to get

OUTPUT:

- If `new=None`, return `True` if this graph allows self-loops or `False` if self-loops are not allowed
- If `new` is a boolean, set the self-loop permission of this graph according to the boolean value of `new`

EXAMPLES:

```
sage: G = Graph()
sage: G._backend.loops()
False
sage: G._backend.loops(True)
sage: G._backend.loops()
True
```

num_edges (*directed*)

Return the number of edges in `self`.

INPUT:

- `directed` – boolean; whether to count `(u, v)` and `(v, u)` as one or two edges

OUTPUT:

- If `directed=True`, counts the number of directed edges in this graph. Otherwise, return the size of this graph.

See also:

- *`num_verts()`* – return the order of this graph.

EXAMPLES:

```
sage: G = Graph(graphs.PetersenGraph())
sage: G._backend.num_edges(False)
15
```

num_verts()

Return the number of vertices in `self`.

OUTPUT:

- The order of this graph.

See also:

- *num_edges()* – return the number of (directed) edges in this graph.

EXAMPLES:

```
sage: G = Graph(graphs.PetersenGraph())
sage: G._backend.num_verts()
10
```

out_degree(*v*)

Return the out-degree of `v`

INPUT:

- `v` – a vertex of the graph.

EXAMPLES:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] }␣
↪)
sage: D.out_degree(1)
2
```

relabel(*perm*, *directed*)

Relabel the graph according to `perm`.

INPUT:

- `perm` – anything which represents a permutation as `v --> perm[v]`, for example a dict or a list
- `directed` – ignored (this is here for compatibility with other backends)

EXAMPLES:

```
sage: G = Graph(graphs.PetersenGraph())
sage: G._backend.relabel(range(9,-1,-1), False)
sage: G.edges()
[(0, 2, None),
 (0, 3, None),
 (0, 5, None),
 (1, 3, None),
 (1, 4, None),
 (1, 6, None),
 (2, 4, None),
 (2, 7, None),
 (3, 8, None),
 (4, 9, None),
 (5, 6, None),
```

(continues on next page)

(continued from previous page)

```
 (5, 9, None),
 (6, 7, None),
 (7, 8, None),
 (8, 9, None)]
```

shortest_path(*x*, *y*, *distance_flag=False*)

Return the shortest path or distance from `x` to `y`.

INPUT:

- `x` – the starting vertex in the shortest path from `x` to `y`
- `y` – the end vertex in the shortest path from `x` to `y`
- `distance_flag` – boolean (default: `False`); when set to `True`, the shortest path distance from `x` to `y` is returned instead of the path

OUTPUT:

- A list of vertices in the shortest path from `x` to `y` or distance from `x` to `y` is returned depending upon the value of parameter `distance_flag`

EXAMPLES:

```
sage: G = Graph(graphs.PetersenGraph())
sage: G.shortest_path(0, 1)
[0, 1]
sage: G.shortest_path_length(0, 1)
1
```

shortest_path_all_vertices(*v*, *cutoff=None*, *distance_flag=False*)

Return for each vertex `u` a shortest `v`–`u` path or distance from `v` to `u`.

INPUT:

- `v` – a starting vertex in the shortest path
- `cutoff` – integer (default: `None`); maximal distance of returned paths (longer paths will not be returned), ignored when set to `None`
- `distance_flag` – boolean (default: `False`); when set to `True`, each vertex `u` connected to `v` is mapped to shortest path distance from `v` to `u` instead of the shortest path in the output dictionary.

OUTPUT:

- A dictionary which maps each vertex `u` connected to `v` to the shortest path list or distance from `v` to `u` depending upon the value of parameter `distance_flag`

Note: The weight of edges is not taken into account.

ALGORITHM:

This is just a breadth-first search.

EXAMPLES:

On the Petersen Graph:

```
sage: g = graphs.PetersenGraph()
sage: paths = g._backend.shortest_path_all_vertices(0)
sage: all((not paths[v] or len(paths[v])-1 == g.distance(0,v)) for v in g)
True
sage: g._backend.shortest_path_all_vertices(0, distance_flag=True)
{0: 0, 1: 1, 2: 2, 3: 2, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2, 9: 2}
```

On a disconnected graph

```
sage: g = 2 * graphs.RandomGNP(20, .3)
sage: paths = g._backend.shortest_path_all_vertices(0)
sage: all((v not in paths and g.distance(0, v) == +Infinity) or len(paths[v])␣
↪- 1 == g.distance(0, v) for v in g)
True
```

shortest_path_special(*x*, *y*, *exclude_vertices=None*, *exclude_edges=None*, *distance_flag=False*)

Return the shortest path or distance from `x` to `y`.

This method is an extension of *shortest_path()* method enabling to exclude vertices and/or edges from the search for the shortest path between `x` and `y`.

INPUT:

- `x` – the starting vertex in the shortest path from `x` to `y`
- `y` – the end vertex in the shortest path from `x` to `y`
- `exclude_vertices` – iterable container (default: `None`); iterable of vertices to exclude from the graph while calculating the shortest path from `x` to `y`
- `exclude_edges` – iterable container (default: `None`); iterable of edges to exclude from the graph while calculating the shortest path from `x` to `y`
- `distance_flag` – boolean (default: `False`); when set to `True`, the shortest path distance from `x` to `y` is returned instead of the path

OUTPUT:

- A list of vertices in the shortest path from `x` to `y` or distance from `x` to `y` is returned depending upon the value of parameter `distance_flag`

EXAMPLES:

```
sage: G = Graph([(1, 2), (2, 3), (3, 4), (1, 5), (5, 6), (6, 7), (7, 4)])
sage: G._backend.shortest_path_special(1, 4)
[1, 2, 3, 4]
sage: G._backend.shortest_path_special(1, 4, exclude_vertices=[5,7])
[1, 2, 3, 4]
sage: G._backend.shortest_path_special(1, 4, exclude_vertices=[2, 3])
[1, 5, 6, 7, 4]
sage: G._backend.shortest_path_special(1, 4, exclude_vertices=[2], exclude_
↪edges=[(5, 6)])
[]
sage: G._backend.shortest_path_special(1, 4, exclude_vertices=[2], exclude_
↪edges=[(2, 3)])
[1, 5, 6, 7, 4]
```

strongly_connected_component_containing_vertex(*v*)

Return the strongly connected component containing the given vertex.

INPUT:

- v – a vertex

EXAMPLES:

The digraph obtained from the PetersenGraph has an unique strongly connected component:

```
sage: g = DiGraph(graphs.PetersenGraph())
sage: g.strongly_connected_component_containing_vertex(0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In the Butterfly DiGraph, each vertex is a strongly connected component:

```
sage: g = digraphs.ButterflyGraph(3)
sage: all([v] == g.strongly_connected_component_containing_vertex(v) for v in␣
↪g)
True
```

class sage.graphs.base.c_graph.**Search_iterator**

Bases: object

An iterator for traversing a (di)graph.

This class is commonly used to perform a depth-first or breadth-first search. The class does not build all at once in memory the whole list of visited vertices. The class maintains the following variables:

- graph – a graph whose vertices are to be iterated over.
- direction – integer; this determines the position at which vertices to be visited are removed from the list stack. For breadth-first search (BFS), element removal occurs at the start of the list, as signified by the value direction=0. This is because in implementations of BFS, the list of vertices to visit are usually maintained by a queue, so element insertion and removal follow a first-in first-out (FIFO) protocol. For depth-first search (DFS), element removal occurs at the end of the list, as signified by the value direction=-1. The reason is that DFS is usually implemented using a stack to maintain the list of vertices to visit. Hence, element insertion and removal follow a last-in first-out (LIFO) protocol.
- stack – a list of vertices to visit
- seen – a list of vertices that are already visited
- test_out – boolean; whether we want to consider the out-neighbors of the graph to be traversed. For undirected graphs, we consider both the in- and out-neighbors. However, for digraphs we only traverse along out-neighbors.
- test_in – boolean; whether we want to consider the in-neighbors of the graph to be traversed. For undirected graphs, we consider both the in- and out-neighbors.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: list(g.breadth_first_search(0))
[0, 1, 4, 5, 2, 6, 3, 9, 7, 8]
```

3.3 Fast sparse graphs

For an overview of graph data structures in sage, see *overview*.

3.3.1 Usage Introduction

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
```

Sparse graphs are initialized as follows:

```
sage: S = SparseGraph(nverts = 10, expected_degree = 3, extra_vertices = 10)
```

This example initializes a sparse graph with room for twenty vertices, the first ten of which are in the graph. In general, the first `nverts` are "active." For example, see that 9 is already in the graph:

```
sage: S.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: S.add_vertex(9)
9
sage: S.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But 10 is not, until we add it:

```
sage: S.add_vertex(10)
10
sage: S.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

You can begin working with unlabeled arcs right away as follows:

```
sage: S.add_arc(0,1)
sage: S.add_arc(1,2)
sage: S.add_arc(1,0)
sage: S.has_arc(7,3)
False
sage: S.has_arc(0,1)
True
sage: S.in_neighbors(1)
[0]
sage: S.out_neighbors(1)
[0, 2]
sage: S.del_all_arcs(0,1)
sage: S.all_arcs(0,1)
[]
sage: S.all_arcs(1,2)
[0]
sage: S.del_vertex(7)
sage: S.all_arcs(7,3)
Traceback (most recent call last):
...
LookupError: vertex (7) is not a vertex of the graph
```

Sparse graphs support multiple edges and labeled edges, but requires that the labels be positive integers (the case label = 0 is treated as no label).

```
sage: S.add_arc_label(0,1,-1)
Traceback (most recent call last):
...
ValueError: Label (-1) must be a nonnegative integer.
sage: S.add_arc(0,1)
```

(continues on next page)

(continued from previous page)

```
sage: S.arc_label(0,1)
0
```

Note that `arc_label` only returns the first edge label found in the specified place, and this can be in any order (if you want all arc labels, use `all_arcs`):

```
sage: S.add_arc_label(0,1,1)
sage: S.arc_label(0,1)
1
sage: S.all_arcs(0,1)
[0, 1]
```

Zero specifies only that there is no labeled arc:

```
sage: S.arc_label(1,2)
0
```

So do not be fooled:

```
sage: S.all_arcs(1,2)
[0]
sage: S.add_arc(1,2)
sage: S.arc_label(1,2)
0
```

Instead, if you work with unlabeled edges, be sure to use the right functions:

```
sage: T = SparseGraph(nverts = 3, expected_degree = 2)
sage: T.add_arc(0,1)
sage: T.add_arc(1,2)
sage: T.add_arc(2,0)
sage: T.has_arc(0,1)
True
```

Sparse graphs are by their nature directed. As of this writing, you need to do operations in pairs to treat the undirected case (or use a backend or a Sage graph):

```
sage: T.has_arc(1,0)
False
```

Multiple unlabeled edges are also possible:

```
sage: for _ in range(10): S.add_arc(5,4)
sage: S.all_arcs(5,4)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The curious developer is encouraged to check out the `unsafe` functions, which do not check input but which run in pure C.

3.3.2 Underlying Data Structure

The class `SparseGraph` contains the following variables which are inherited from `CGraph` (for explanation, refer to the documentation there):

```
cdef int num_verts
cdef int num_arcs
cdef int *in_degrees
cdef int *out_degrees
cdef bitset_t active_vertices
```

It also contains the following variables:

```
cdef int hash_length
cdef int hash_mask
cdef SparseGraphBTNode **vertices
```

For each vertex `u`, a hash table of length `hash_length` is instantiated. An arc `(u, v)` is stored at `u * hash_length + hash(v)` of the array `vertices`, where `hash` should be thought of as an arbitrary but fixed hash function which takes values in `0 <= hash < hash_length`. Each address may represent different arcs, say `(u, v1)` and `(u, v2)` where `hash(v1) == hash(v2)`. Thus, a binary tree structure is used at this step to speed access to individual arcs, whose nodes (each of which represents a pair `(u,v)`) are instances of the following type:

```
cdef struct SparseGraphBTNode:
    int vertex
    int number
    SparseGraphLLNode *labels
    SparseGraphBTNode *left
    SparseGraphBTNode *right
```

Which range of the `vertices` array the root of the tree is in determines `u`, and `vertex` stores `v`. The integer `number` stores only the number of unlabeled arcs from `u` to `v`.

Currently, labels are stored in a simple linked list, whose nodes are instances of the following type:

```
cdef struct SparseGraphLLNode:
    int label
    int number
    SparseGraphLLNode *next
```

The int `label` must be a positive integer, since 0 indicates no label, and negative numbers indicate errors. The int `number` is the number of arcs with the given label.

TODO: Optimally, edge labels would also be represented by a binary tree, which would help performance in graphs with many overlapping edges. Also, a more efficient binary tree structure could be used, although in practice the trees involved will usually have very small order, unless the degree of vertices becomes significantly larger than the `expected_degree` given, because this is the size of each hash table. Indeed, the expected size of the binary trees is $\frac{\text{actual degree}}{\text{expected degree}}$. Ryan Dingman, e.g., is working on a general-purpose Cython-based red black tree, which would be optimal for both of these uses.

class sage.graphs.base.sparse_graph.**SparseGraph**

Bases: *sage.graphs.base.c_graph.CGraph*

Compiled sparse graphs.

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
```

Sparse graphs are initialized as follows:

```
sage: S = SparseGraph(nverts = 10, expected_degree = 3, extra_vertices = 10)
```

INPUT:

- `nverts` - non-negative integer, the number of vertices.
- **`expected_degree` - non-negative integer (default: 16), expected upper** bound on degree of vertices.
- **`extra_vertices` - non-negative integer (default: 0), how many extra** vertices to allocate.
- `verts` - optional list of vertices to add
- `arcs` - optional list of arcs to add

The first `nverts` are created as vertices of the graph, and the next `extra_vertices` can be freely added without reallocation. See top level documentation for more details. The input `verts` and `arcs` are mainly for use in pickling.

`add_arc_label`(*u*, *v*, *l=0*)

Adds arc `(u, v)` to the graph with label `l`.

INPUT:

- `u, v` - non-negative integers, must be in self
- `l` - a positive integer label, or zero for no label

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(0,1)
sage: G.add_arc_label(4,7)
Traceback (most recent call last):
...
LookupError: vertex (7) is not a vertex of the graph
sage: G.has_arc(1,0)
False
sage: G.has_arc(0,1)
True
sage: G.add_arc_label(1,2,2)
sage: G.arc_label(1,2)
2
```

`all_arcs`(*u*, *v*)

Gives the labels of all arcs `(u, v)`. An unlabeled arc is interpreted as having label 0.

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(1,2,1)
sage: G.add_arc_label(1,2,2)
sage: G.add_arc_label(1,2,2)
sage: G.add_arc_label(1,2,2)
sage: G.add_arc_label(1,2,3)
sage: G.add_arc_label(1,2,3)
sage: G.add_arc_label(1,2,4)
sage: G.all_arcs(1,2)
[4, 3, 3, 2, 2, 2, 1]
```

`arc_label`(*u*, *v*)

Retrieves the first label found associated with `(u, v)`.

INPUT:

- u, v - non-negative integers, must be in self

OUTPUT:

- positive integer - indicates that there is a label on (u, v).
- 0 - either the arc (u, v) is unlabeled, or there is no arc at all.

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(3,4,7)
sage: G.arc_label(3,4)
7
```

To this function, an unlabeled arc is indistinguishable from a non-arc:

```
sage: G.add_arc_label(1,0)
sage: G.arc_label(1,0)
0
sage: G.arc_label(1,1)
0
```

This function only returns the *first* label it finds from u to v:

```
sage: G.add_arc_label(1,2,1)
sage: G.add_arc_label(1,2,2)
sage: G.arc_label(1,2)
2
```

del_arc_label(*u*, *v*, *l*)

Delete an arc (u, v) with label l.

INPUT:

- u, v - non-negative integers, must be in self
- l - a positive integer label, or zero for no label

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(0,1,0)
sage: G.add_arc_label(0,1,1)
sage: G.add_arc_label(0,1,2)
sage: G.add_arc_label(0,1,2)
sage: G.add_arc_label(0,1,3)
sage: G.del_arc_label(0,1,2)
sage: G.all_arcs(0,1)
[0, 3, 2, 1]
sage: G.del_arc_label(0,1,0)
sage: G.all_arcs(0,1)
[3, 2, 1]
```

has_arc_label(*u*, *v*, *l*)

Indicates whether there is an arc (u, v) with label l.

INPUT:

- u, v – non-negative integers, must be in self

- `l` – a positive integer label, or zero for no label

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc_label(0,1,0)
sage: G.add_arc_label(0,1,1)
sage: G.add_arc_label(0,1,2)
sage: G.add_arc_label(0,1,2)
sage: G.has_arc_label(0,1,1)
True
sage: G.has_arc_label(0,1,2)
True
sage: G.has_arc_label(0,1,3)
False
```

in_degree(*u*)

Returns the in-degree of `v`

INPUT:

- `u` - integer

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc(0,1)
sage: G.add_arc(1,2)
sage: G.add_arc(1,3)
sage: G.in_degree(0)
0
sage: G.in_degree(1)
1
```

out_degree(*u*)

Returns the out-degree of `v`

INPUT:

- `u` - integer

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(5)
sage: G.add_arc(0,1)
sage: G.add_arc(1,2)
sage: G.add_arc(1,3)
sage: G.out_degree(0)
1
sage: G.out_degree(1)
2
```

realloc(*total*)

Reallocate the number of vertices to use, without actually adding any.

INPUT:

- `total` - integer, the total size to make the array

Returns -1 and fails if reallocation would destroy any active vertices.

EXAMPLES:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: S = SparseGraph(nverts=4, extra_vertices=4)
sage: S.current_allocation()
8
sage: S.add_vertex(6)
6
sage: S.current_allocation()
8
sage: S.add_vertex(10)
10
sage: S.current_allocation()
16
sage: S.add_vertex(40)
Traceback (most recent call last):
...
RuntimeError: requested vertex is past twice the allocated range: use realloc
sage: S.realloc(50)
sage: S.add_vertex(40)
40
sage: S.current_allocation()
50
sage: S.realloc(30)
-1
sage: S.current_allocation()
50
sage: S.del_vertex(40)
sage: S.realloc(30)
sage: S.current_allocation()
30
```

class sage.graphs.base.sparse_graph.**SparseGraphBackend**

Bases: *sage.graphs.base.c_graph.CGraphBackend*

Backend for Sage graphs using SparseGraphs.

```
sage: from sage.graphs.base.sparse_graph import SparseGraphBackend
```

This class is only intended for use by the Sage Graph and DiGraph class. If you are interested in using a SparseGraph, you probably want to do something like the following example, which creates a Sage Graph instance which wraps a SparseGraph object:

```
sage: G = Graph(30, sparse=True)
sage: G.add_edges([(0,1), (0,3), (4,5), (9, 23)])
sage: G.edges(labels=False)
[(0, 1), (0, 3), (4, 5), (9, 23)]
```

Note that Sage graphs using the backend are more flexible than SparseGraphs themselves. This is because SparseGraphs (by design) do not deal with Python objects:

```
sage: G.add_vertex((0,1,2))
sage: sorted(list(G),
....:        key=lambda x: (isinstance(x, tuple), x))
[0,
...
```

(continues on next page)

(continued from previous page)

```
 29,
 (0, 1, 2)]
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: SG = SparseGraph(30)
sage: SG.add_vertex((0,1,2))
Traceback (most recent call last):
...
TypeError: an integer is required
```

add_edge(*u*, *v*, *l*, *directed*)

Adds the edge `(u,v)` to self.

INPUT:

- `u,v` - the vertices of the edge
- `l` - the edge label
- `directed` - if False, also add `(v,u)`

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edge(0,1,None,False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None)]
```

add_edges(*edges*, *directed*)

Add edges from a list.

INPUT:

- `edges` - the edges to be added - can either be of the form `(u,v)` or `(u,v,l)`
- `directed` - if False, add `(v,u)` as well as `(u,v)`

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edges([(0,1), (2,3), (4,5), (5,6)], False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None),
 (2, 3, None),
 (4, 5, None),
 (5, 6, None)]
```

del_edge(*u*, *v*, *l*, *directed*)

Delete edge `(u,v,l)`.

INPUT:

- `u,v` - the vertices of the edge
- `l` - the edge label
- `directed` - if False, also delete `(v,u,l)`

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edges([(0,1), (2,3), (4,5), (5,6)], False)
```

(continues on next page)

(continued from previous page)

```
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None),
 (2, 3, None),
 (4, 5, None),
 (5, 6, None)]
sage: D.del_edge(0,1,None,True)
sage: list(D.iterator_out_edges(range(9), True))
[(1, 0, None),
 (2, 3, None),
 (3, 2, None),
 (4, 5, None),
 (5, 4, None),
 (5, 6, None),
 (6, 5, None)]
```

get_edge_label(*u*, *v*)

Returns the edge label for `(u,v)`.

INPUT:

- `u,v` - the vertices of the edge

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edges([(0,1,1), (2,3,2), (4,5,3), (5,6,2)], False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, 1), (2, 3, 2), (4, 5, 3), (5, 6, 2)]
sage: D.get_edge_label(3,2)
2
```

has_edge(*u*, *v*, *l*)

Returns whether this graph has edge `(u,v)` with label `l`. If `l` is `None`, return whether this graph has an edge `(u,v)` with any label.

INPUT:

- `u,v` - the vertices of the edge
- `l` - the edge label, or `None`

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edges([(0,1), (2,3), (4,5), (5,6)], False)
sage: D.has_edge(0,1,None)
True
```

iterator_edges(*vertices*, *labels*)

Iterate over the edges incident to a sequence of vertices.

Edges are assumed to be undirected.

Warning: This will try to sort the two ends of every edge.

INPUT:

- `vertices` – a list of vertex labels

- `labels` – boolean, whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.add_edge(1,2,3,False)
sage: list(G.iterator_edges(range(9), False))
[(1, 2)]
sage: list(G.iterator_edges(range(9), True))
[(1, 2, 3)]
```

iterator_in_edges (*vertices*, *labels*)
Iterate over the incoming edges incident to a sequence of vertices.

INPUT:

- `vertices` - a list of vertex labels
- `labels` - boolean, whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.add_edge(1,2,3,True)
sage: list(G.iterator_in_edges([1], False))
[]
sage: list(G.iterator_in_edges([2], False))
[(1, 2)]
sage: list(G.iterator_in_edges([2], True))
[(1, 2, 3)]
```

iterator_out_edges (*vertices*, *labels*)
Iterate over the outbound edges incident to a sequence of vertices.

INPUT:

- `vertices` - a list of vertex labels
- `labels` - boolean, whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.add_edge(1,2,3,True)
sage: list(G.iterator_out_edges([2], False))
[]
sage: list(G.iterator_out_edges([1], False))
[(1, 2)]
sage: list(G.iterator_out_edges([1], True))
[(1, 2, 3)]
```

iterator_unsorted_edges (*vertices*, *labels*)
Iterate over the edges incident to a sequence of vertices.

Edges are assumed to be undirected.

This does not sort the ends of each edge.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean, whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.add_edge(1,2,3,False)
sage: list(G.iterator_unsorted_edges(range(9), False))
[(2, 1)]
sage: list(G.iterator_unsorted_edges(range(9), True))
[(2, 1, 3)]
```

multiple_edges (*new*)

Get/set whether or not `self` allows multiple edges.

INPUT:

- `new` - boolean (to set) or `None` (to get)

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.multiple_edges(True)
sage: G.multiple_edges(None)
True
sage: G.multiple_edges(False)
sage: G.multiple_edges(None)
False
sage: G.add_edge(0,1,0,True)
sage: G.add_edge(0,1,0,True)
sage: list(G.iterator_edges(range(9), True))
[(0, 1, 0)]
```

set_edge_label (*u*, *v*, *l*, *directed*)

Label the edge `(u,v)` by `l`.

INPUT:

- `u,v` - the vertices of the edge
- `l` - the edge label
- `directed` - if False, also set `(v,u)` with label `l`

EXAMPLES:

```
sage: G = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: G.add_edge(1,2,None,True)
sage: G.set_edge_label(1,2,'a',True)
sage: list(G.iterator_edges(range(9), True))
[(1, 2, 'a')]
```

Note that it fails silently if there is no edge there:

```
sage: G.set_edge_label(2,1,'b',True)
sage: list(G.iterator_edges(range(9), True))
[(1, 2, 'a')]
```

3.4 Fast dense graphs

For an overview of graph data structures in sage, see *overview*.

3.4.1 Usage Introduction

```
sage: from sage.graphs.base.dense_graph import DenseGraph
```

Dense graphs are initialized as follows:

```
sage: D = DenseGraph(nverts=10, extra_vertices=10)
```

This example initializes a dense graph with room for twenty vertices, the first ten of which are in the graph. In general, the first `nverts` are "active." For example, see that 9 is already in the graph:

```
sage: D.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: D.add_vertex(9)
9
sage: D.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But 10 is not, until we add it:

```
sage: D.add_vertex(10)
10
sage: D.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

You can begin working right away as follows:

```
sage: D.add_arc(0, 1)
sage: D.add_arc(1, 2)
sage: D.add_arc(1, 0)
sage: D.has_arc(7, 3)
False
sage: D.has_arc(0, 1)
True
sage: D.in_neighbors(1)
[0]
sage: D.out_neighbors(1)
[0, 2]
sage: D.del_all_arcs(0, 1)
sage: D.has_arc(0, 1)
False
sage: D.has_arc(1, 2)
True
sage: D.del_vertex(7)
sage: D.has_arc(7, 3)
False
```

Dense graphs do not support multiple or labeled edges.

```
sage: T = DenseGraph(nverts=3, extra_vertices=2)
sage: T.add_arc(0, 1)
sage: T.add_arc(1, 2)
sage: T.add_arc(2, 0)
sage: T.has_arc(0, 1)
True
```

```
sage: for _ in range(10): D.add_arc(5, 4)
sage: D.has_arc(5,4 )
True
```

Dense graphs are by their nature directed. As of this writing, you need to do operations in pairs to treat the undirected case (or use a backend or a Sage graph):

```
sage: T.has_arc(1, 0)
False
```

The curious developer is encouraged to check out the `unsafe` functions, which do not check input but which run in pure C.

3.4.2 Underlying Data Structure

The class `DenseGraph` contains the following variables which are inherited from `CGraph` (for explanation, refer to the documentation there):

```
cdef int num_verts
cdef int num_arcs
cdef int *in_degrees
cdef int *out_degrees
cdef bitset_t active_vertices
```

It also contains the following variables:

```
cdef int num_longs
cdef unsigned long *edges
```

The array `edges` is a series of bits which are turned on or off, and due to this, dense graphs only support graphs without edge labels and with no multiple edges. `num_longs` stores the length of the `edges` array. Recall that this length reflects the number of available vertices, not the number of "actual" vertices. For more details about this, refer to the documentation for `CGraph`.

class sage.graphs.base.dense_graph.**DenseGraph**

Bases: *sage.graphs.base.c_graph.CGraph*

Compiled dense graphs.

```
sage: from sage.graphs.base.dense_graph import DenseGraph
```

Dense graphs are initialized as follows:

```
sage: D = DenseGraph(nverts=10, extra_vertices=10)
```

INPUT:

- `nverts` – non-negative integer; the number of vertices
- `extra_vertices` – non-negative integer (default: 10); how many extra vertices to allocate
- `verts` – list (default: `None`); optional list of vertices to add
- `arcs` – list (default: `None`); optional list of arcs to add

The first `nverts` are created as vertices of the graph, and the next `extra_vertices` can be freely added without reallocation. See top level documentation for more details. The input `verts` and `arcs` are mainly for use in pickling.

complement()
Replace the graph with its complement

Note: Assumes that the graph has no loop.

EXAMPLES:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(5)
sage: G.add_arc(0, 1)
sage: G.has_arc(0, 1)
True
sage: G.complement()
sage: G.has_arc(0, 1)
False
```

realloc(*total_verts*)
Reallocate the number of vertices to use, without actually adding any.

INPUT:

- `total` – integer; the total size to make the array

Returns -1 and fails if reallocation would destroy any active vertices.

EXAMPLES:

```
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: D = DenseGraph(nverts=4, extra_vertices=4)
sage: D.current_allocation()
8
sage: D.add_vertex(6)
6
sage: D.current_allocation()
8
sage: D.add_vertex(10)
10
sage: D.current_allocation()
16
sage: D.add_vertex(40)
Traceback (most recent call last):
...
RuntimeError: requested vertex is past twice the allocated range: use realloc
sage: D.realloc(50)
sage: D.add_vertex(40)
40
sage: D.current_allocation()
50
sage: D.realloc(30)
-1
sage: D.current_allocation()
50
sage: D.del_vertex(40)
sage: D.realloc(30)
sage: D.current_allocation()
30
```

class sage.graphs.base.dense_graph.**DenseGraphBackend**

Bases: *sage.graphs.base.c_graph.CGraphBackend*

Backend for Sage graphs using DenseGraphs.

```
sage: from sage.graphs.base.dense_graph import DenseGraphBackend
```

This class is only intended for use by the Sage Graph and DiGraph class. If you are interested in using a DenseGraph, you probably want to do something like the following example, which creates a Sage Graph instance which wraps a `DenseGraph` object:

```
sage: G = Graph(30, sparse=False)
sage: G.add_edges([(0, 1), (0, 3), (4, 5), (9, 23)])
sage: G.edges(labels=False)
[(0, 1), (0, 3), (4, 5), (9, 23)]
```

Note that Sage graphs using the backend are more flexible than DenseGraphs themselves. This is because DenseGraphs (by design) do not deal with Python objects:

```
sage: G.add_vertex((0, 1, 2))
sage: sorted(list(G),
....:        key=lambda x: (isinstance(x, tuple), x))
[0,
...
 29,
 (0, 1, 2)]
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: DG = DenseGraph(30)
sage: DG.add_vertex((0, 1, 2))
Traceback (most recent call last):
...
TypeError: an integer is required
```

add_edge(*u*, *v*, *l*, *directed*)

Add edge `(u, v)` to self.

INPUT:

- `u,v` – the vertices of the edge
- `l` – the edge label (ignored)
- `directed` – if `False`, also add `(v, u)`

Note: The input `l` is for consistency with other backends.

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_edge(0, 1, None, False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None)]
```

add_edges(*edges*, *directed*)

Add edges from a list.

INPUT:

- **edges – an iterable of edges to be added; each edge can either be** of the form `(u, v)` or `(u, v, l)`

- directed – if False, adds (v, u) as well as (u, v)

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_edges([(0, 1), (2, 3), (4, 5), (5, 6)], False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None),
 (2, 3, None),
 (4, 5, None),
 (5, 6, None)]
```

del_edge(*u*, *v*, *l*, *directed*)

Delete edge (u, v).

INPUT:

- u, v – the vertices of the edge
- l – the edge label (ignored)
- directed – if False, also delete (v, u, l)

Note: The input l is for consistency with other backends.

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_edges([(0, 1), (2, 3), (4, 5), (5, 6)], False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None),
 (2, 3, None),
 (4, 5, None),
 (5, 6, None)]
sage: D.del_edge(0, 1, None, True)
sage: list(D.iterator_out_edges(range(9), True))
[(1, 0, None),
 (2, 3, None),
 (3, 2, None),
 (4, 5, None),
 (5, 4, None),
 (5, 6, None),
 (6, 5, None)]
```

get_edge_label(*u*, *v*)

Return the edge label for (u, v).

Always None, since dense graphs do not support edge labels.

INPUT:

- u, v – the vertices of the edge

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_edges([(0, 1), (2, 3, 7), (4, 5), (5, 6)], False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None),
 (2, 3, None),
```

(continues on next page)

(continued from previous page)

```
 (4, 5, None),
 (5, 6, None)]
sage: D.del_edge(0, 1, None, True)
sage: list(D.iterator_out_edges(range(9), True))
[(1, 0, None),
 (2, 3, None),
 (3, 2, None),
 (4, 5, None),
 (5, 4, None),
 (5, 6, None),
 (6, 5, None)]
sage: D.get_edge_label(2, 3)
sage: D.get_edge_label(2, 4)
Traceback (most recent call last):
...
LookupError: (2, 4) is not an edge of the graph
```

has_edge(*u*, *v*, *l*)

Check whether this graph has edge `(u, v)`.

Note: The input `l` is for consistency with other backends.

INPUT:

- `u,v` – the vertices of the edge
- `l` – the edge label (ignored)

EXAMPLES:

```
sage: D = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: D.add_edges([(0, 1), (2, 3), (4, 5), (5, 6)], False)
sage: D.has_edge(0, 1, None)
True
```

iterator_edges(*vertices*, *labels*)

Return an iterator over the edges incident to a sequence of vertices.

Edges are assumed to be undirected.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean; whether to return edge labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: G.add_edge(1, 2, None, False)
sage: list(G.iterator_edges(range(9), False))
[(1, 2)]
sage: list(G.iterator_edges(range(9), True))
[(1, 2, None)]
```

iterator_in_edges(*vertices*, *labels*)

Return an iterator over the incoming edges incident to a sequence of vertices.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean; whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: G.add_edge(1, 2, None, True)
sage: list(G.iterator_in_edges([1], False))
[]
sage: list(G.iterator_in_edges([2], False))
[(1, 2)]
sage: list(G.iterator_in_edges([2], True))
[(1, 2, None)]
```

iterator_out_edges(*vertices*, *labels*)

Return an iterator over the outbound edges incident to a sequence of vertices.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean; whether to return labels as well

EXAMPLES:

```
sage: G = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: G.add_edge(1, 2, None, True)
sage: list(G.iterator_out_edges([2], False))
[]
sage: list(G.iterator_out_edges([1], False))
[(1, 2)]
sage: list(G.iterator_out_edges([1], True))
[(1, 2, None)]
```

multiple_edges(*new*)

Get/set whether or not `self` allows multiple edges.

INPUT:

- `new` – boolean (to set) or `None` (to get)

EXAMPLES:

```
sage: import sage.graphs.base.dense_graph
sage: G = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: G.multiple_edges(True)
Traceback (most recent call last):
...
NotImplementedError: dense graphs do not support multiple edges
sage: G.multiple_edges(None)
False
```

set_edge_label(*u*, *v*, *l*, *directed*)

Label the edge `(u, v)` by `l`.

INPUT:

- `u,v` – the vertices of the edge
- `l` – the edge label
- `directed` – if `False`, also set `(v, u)` with label `l`

EXAMPLES:

```
sage: import sage.graphs.base.dense_graph
sage: G = sage.graphs.base.dense_graph.DenseGraphBackend(9)
sage: G.set_edge_label(1, 2, 'a', True)
Traceback (most recent call last):
...
NotImplementedError: dense graphs do not support edge labels
```

3.5 Static dense graphs

This module gathers everything which is related to static dense graphs, i.e. :

- The vertices are integer from 0 to $n-1$
- No labels on vertices/edges
- No multiple edges
- No addition/removal of vertices

This being said, it is technically possible to add/remove edges. The data structure does not mind at all.

It is all based on the binary matrix data structure described in `misc/binary_matrix.pxi`, which is almost a copy of the bitset data structure. The only difference is that it differentiates the rows (the vertices) instead of storing the whole data in a long bitset, and we can use that.

For an overview of graph data structures in sage, see *overview*.

3.5.1 Index

Cython functions

`dense_graph_init`	Fill a binary matrix with the information from a Sage (di)graph.

Python functions

is_strongly_regular()	Check whether the graph is strongly regular
is_triangle_free()	Check whether G is triangle free
triangles_count()	Return the number of triangles containing v, for every v
connected_subgraph_iterator()	Iterator over the induced connected subgraphs of order at most k

3.5.2 Functions

`sage.graphs.base.static_dense_graph.`**`connected_subgraph_iterator`**(*G*, *k=None*, *vertices_only=False*)

Iterator over the induced connected subgraphs of order at most k.

This method implements a iterator over the induced connected subgraphs of the input (di)graph. An induced subgraph of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset (Wikipedia article Induced_subgraph).

As for method `sage.graphs.generic_graph.connected_components()`, edge orientation is ignored. Hence, the directed graph with a single arc $0 \to 1$ is considered connected.

INPUT:

- `G` – a *Graph* or a *DiGraph*; loops and multiple edges are allowed
- `k` – (optional) integer; maximum order of the connected subgraphs to report; by default, the method iterates over all connected subgraphs (equivalent to `k == n`)
- `vertices_only` – boolean (default: `False`); whether to return (Di)Graph or list of vertices

EXAMPLES:

```
sage: G = DiGraph([(1, 2), (2, 3), (3, 4), (4, 2)])
sage: list(G.connected_subgraph_iterator())
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 4 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 3 vertices,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(vertices_only=True))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 4],
 [2], [2, 3], [2, 3, 4], [2, 4], [3], [3, 4], [4]]
sage: list(G.connected_subgraph_iterator(k=2))
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(k=2, vertices_only=True))
[[1], [1, 2], [2], [2, 3], [2, 4], [3], [3, 4], [4]]

sage: G = DiGraph([(1, 2), (2, 1)])
sage: list(G.connected_subgraph_iterator())
[Subgraph of (): Digraph on 1 vertex,
 Subgraph of (): Digraph on 2 vertices,
 Subgraph of (): Digraph on 1 vertex]
sage: list(G.connected_subgraph_iterator(vertices_only=True))
[[1], [1, 2], [2]]
```

`sage.graphs.base.static_dense_graph.`**`is_strongly_regular`**(*g*, *parameters=False*)

Check whether the graph is strongly regular.

A simple graph G is said to be strongly regular with parameters (n, k, λ, μ) if and only if:

- G has n vertices
- G is k-regular
- Any two adjacent vertices of G have λ common neighbors

- Any two non-adjacent vertices of G have μ common neighbors

By convention, the complete graphs, the graphs with no edges and the empty graph are not strongly regular.

See the Wikipedia article Strongly regular graph.

INPUT:

- `parameters` – boolean (default: `False`); whether to return the quadruple (n, k, λ, μ). If `parameters = False` (default), this method only returns `True` and `False` answers. If `parameters = True`, the `True` answers are replaced by quadruples (n, k, λ, μ). See definition above.

EXAMPLES:

Petersen's graph is strongly regular:

```
sage: g = graphs.PetersenGraph()
sage: g.is_strongly_regular()
True
sage: g.is_strongly_regular(parameters=True)
(10, 3, 0, 1)
```

And Clebsch's graph is too:

```
sage: g = graphs.ClebschGraph()
sage: g.is_strongly_regular()
True
sage: g.is_strongly_regular(parameters=True)
(16, 5, 0, 2)
```

But Chvatal's graph is not:

```
sage: g = graphs.ChvatalGraph()
sage: g.is_strongly_regular()
False
```

Complete graphs are not strongly regular. (trac ticket #14297)

```
sage: g = graphs.CompleteGraph(5)
sage: g.is_strongly_regular()
False
```

Completements of complete graphs are not strongly regular:

```
sage: g = graphs.CompleteGraph(5).complement()
sage: g.is_strongly_regular()
False
```

The empty graph is not strongly regular:

```
sage: g = graphs.EmptyGraph()
sage: g.is_strongly_regular()
False
```

If the input graph has loops or multiedges an exception is raised:

```
sage: Graph([(1,1),(2,2)],loops=True).is_strongly_regular()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with
```

(continues on next page)

(continued from previous page)

```
loops. Perhaps this method can be updated to handle them, but in the
meantime if you want to use it please disallow loops using
allow_loops().

sage: Graph([(1,2),(1,2)],multiedges=True).is_strongly_regular()
Traceback (most recent call last):
...
ValueError: This method is not known to work on graphs with
multiedges. Perhaps this method can be updated to handle them, but in
the meantime if you want to use it please disallow multiedges using
allow_multiple_edges().
```

sage.graphs.base.static_dense_graph.**is_triangle_free**(*G*, *certificate=False*)

Check whether G is triangle free.

INPUT:

- `G` – a Sage graph
- `certificate` – boolean (default: `False`); whether to return a triangle if one is found

EXAMPLES:

```
sage: from sage.graphs.base.static_dense_graph import is_triangle_free
sage: is_triangle_free(graphs.PetersenGraph())
True
sage: K4 = graphs.CompleteGraph(4)
sage: is_triangle_free(K4)
False
sage: b, certif = is_triangle_free(K4, certificate=True)
sage: K4.subgraph(certif).is_clique()
True
```

sage.graphs.base.static_dense_graph.**triangles_count**(*G*)

Return the number of triangles containing v, for every v.

INPUT:

- `G` – a simple Sage graph

EXAMPLES:

```
sage: from sage.graphs.base.static_dense_graph import triangles_count
sage: triangles_count(graphs.PetersenGraph())
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0}
sage: sum(triangles_count(graphs.CompleteGraph(15)).values()) == 3 * binomial(15,␣
↪3)
True
```

3.6 Static Sparse Graphs

3.6.1 What is the point ?

This class implements a Cython (di)graph structure made for efficiency. The graphs are *static*, i.e. no add/remove vertex/edges methods are available, nor can they easily or efficiently be implemented within this data structure.

The data structure, however, is made to save the maximum amount of computations for graph algorithms whose main operation is to *list the out-neighbours of a vertex* (which is precisely what BFS, DFS, distance computations and the flow-related stuff waste their life on).

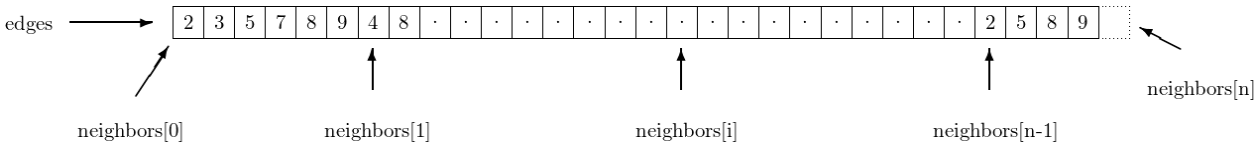
The code contained in this module is written C-style. The purpose is efficiency and simplicity.

For an overview of graph data structures in sage, see *overview*.

Author:

- Nathann Cohen (2011)

3.6.2 Data structure



The data structure is actually pretty simple and compact. `short_digraph` has five fields

- `n` (`int`); the number of vertices in the graph
- `m` (`int`); the number of edges in the graph
- `edges` (`uint32_t *`); array whose length is the number of edges of the graph
- `neighbors` (`uint32_t **`); this array has size $n + 1$, and describes how the data of `edges` should be read : the neighbors of vertex i are the elements of `edges` addressed by `neighbors[i]... neighbors[i+1]-1`. The element `neighbors[n]`, which corresponds to no vertex (they are numbered from 0 to $n - 1$) is present so that it remains easy to enumerate the neighbors of vertex $n - 1$: the last of them is the element addressed by `neighbors[n]-1`.
- `edge_labels` (`list`); this cython list associates a label to each edge of the graph. If a given edge is represented by `edges[i]`, this its associated label can be found at `edge_labels[i]`. This object is usually NULL, unless the call to `init_short_digraph` explicitly requires the labels to be stored in the data structure.

In the example given above, vertex 0 has 2,3,5,7,8 and 9 as out-neighbors, but not 4, which is an out-neighbour of vertex 1. Vertex $n - 1$ has 2, 5, 8 and 9 as out-neighbors. `neighbors[n]` points toward the cell immediately *after* the end of `edges`, hence *outside of the allocated memory*. It is used to indicate the end of the outneighbors of vertex $n - 1$

Iterating over the edges

This is *the one thing* to have in mind when working with this data structure:

```
cdef list_edges(short_digraph g):
    cdef int i, j
    for i in range(g.n):
        for j in range(g.neighbors[i+1]-g.neighbors[i]):
            print("There is an edge from {} to {}".format(i, g.neighbors[i][j]))
```

Advantages

Two great points :

- The neighbors of a vertex are C types, and are contiguous in memory.
- Storing such graphs is incredibly cheaper than storing Python structures.

Well, I think it would be hard to have anything more efficient than that to enumerate out-neighbors in sparse graphs ! :-)

3.6.3 Technical details

- When creating a `short_digraph` from a `Graph` or `DiGraph` named `G`, the i^{th} vertex corresponds *by default* to `G.vertices()[i]`. Using optional parameter `vertex_list`, you can specify the order of the vertices. Then i^{th} vertex will corresponds to `vertex_list[i]`.
- Some methods return `bitset_t` objets when lists could be expected. There is a very useful `bitset_list` function for this kind of problems :-)
- When the edges are labelled, most of the space taken by this graph is taken by edge labels. If no edge is labelled then this space is not allocated, but if *any* edge has a label then a (possibly empty) label is stored for each edge, which can double the memory needs.
- The data structure stores the number of edges, even though it appears that this number can be reconstructed with `g.neighbors[n]-g.neighbors[0]`. The trick is that not all elements of the `g.edges` array are necessarily used : when an undirected graph contains loops, only one entry of the array of size $2m$ is used to store it, instead of the expected two. Storing the number of edges is the only way to avoid an uselessly costly computation to obtain the number of edges of an undirected, looped, AND labelled graph (think of several loops on the same vertex with different labels).
- The codes of this module are well documented, and many answers can be found directly in the code.

3.6.4 Cython functions

`init_short_digraph(short_digraph g, G)`	Initialize `short_digraph g` from a Sage (Di)Graph.
`int n_edges(short_digraph g)`	Return the number of edges in `g`
`int out_degree(short_digraph g, int i)`	Return the out-degree of vertex i in `g`
`has_edge(short_digraph g, int u, int v)`	Test the existence of an edge.
`edge_label(short_digraph g, int * edge)`	Return the label associated with a given edge
`init_empty_copy(short_digraph dst, short_digraph src)`	Allocate `dst` so that it can contain as many vertices and edges as `src`.
`init_reverse(short_digraph dst, short_digraph src)`	Initialize `dst` to a copy of `src` with all edges in the opposite direction.
`free_short_digraph(short_digraph g)`	Free the resources used by `g`

Connectivity

`can_be_reached_from(short_digraph g, int src, bitset_t reached)`

Assuming `bitset_t reached` has size at least `g.n`, this method updates `reached` so that it represents the set of vertices that can be reached from `src` in `g`.

`strongly_connected_component_containing_vertex(short_digraph g, short_digraph g_reversed, int v, bitset_t scc)`

Assuming `bitset_t reached` has size at least `g.n`, this method updates `scc` so that it represents the vertices of the strongly connected component containing `v` in `g`. The variable `g_reversed` is assumed to represent the reverse of `g`.

`tarjan_strongly_connected_components_C(short_digraph g, int *scc)`

Assuming `scc` is already allocated and has size at least `g.n`, this method computes the strongly connected components of `g`, and outputs in `scc[v]` the number of the strongly connected component containing `v`. It returns the number of strongly connected components.

`strongly_connected_components_digraph_C(short_digraph g, int nscc, int *scc, short_digraph output):`

Assuming `nscc` and `scc` are the outputs of `tarjan_strongly_connected_components_C` on `g`, this routine sets `output` to the strongly connected component digraph of `g`, that is, the vertices of `output` are the strongly connected components of `g` (numbers are provided by `scc`), and `output` contains an arc `(C1,C2)` if `g` has an arc from a vertex in `C1` to a vertex in `C2`.

3.6.5 What is this module used for ?

It is for instance used in the *sage.graphs.distances_all_pairs* module, and in the *strongly_connected_components()* method.

3.6.6 Python functions

These functions are available so that Python modules from Sage can call the Cython routines this module implements (as they can not directly call methods with C arguments).

`sage.graphs.base.static_sparse_graph.`**`spectral_radius`**(*G*, *prec=1e-10*)

Return an interval of floating point number that encloses the spectral radius of this graph

The input graph `G` must be *strongly connected*.

INPUT:

- `prec` – (default `1e-10`) an upper bound for the relative precision of the interval

The algorithm is iterative and uses an inequality valid for non-negative matrices. Namely, if A is a non-negative square matrix with Perron-Frobenius eigenvalue λ then the following inequality is valid for any vector x

$$\min_i \frac{(Ax)_i}{x_i} \leq \lambda \leq \max_i \frac{(Ax)_i}{x_i}$$

Note: The speed of convergence of the algorithm is governed by the spectral gap (the distance to the second largest modulus of other eigenvalues). If this gap is small, then this function might not be appropriate.

The algorithm is not smart and not parallel! It uses basic interval arithmetic and native floating point arithmetic.

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import spectral_radius

sage: G = DiGraph([(0,0),(0,1),(1,0)], loops=True)
```

(continues on next page)

(continued from previous page)

```
sage: phi = (RR(1) + RR(5).sqrt() ) / 2
sage: phi  # abs tol 1e-14
1.618033988749895
sage: e_min, e_max = spectral_radius(G, 1e-14)
sage: e_min, e_max     # abs tol 1e-14
(1.618033988749894, 1.618033988749896)
sage: (e_max - e_min)  # abs tol 1e-14
1e-14
sage: e_min < phi < e_max
True
```

This function also works for graphs:

```
sage: G = Graph([(0,1),(0,2),(1,2),(1,3),(2,4),(3,4)])
sage: e_min, e_max = spectral_radius(G, 1e-14)
sage: e = max(G.adjacency_matrix().charpoly().roots(AA, multiplicities=False))
sage: e_min < e < e_max
True

sage: G.spectral_radius()  # abs tol 1e-9
(2.48119430408, 2.4811943041)
```

A larger example:

```
sage: G = DiGraph()
sage: G.add_edges((i,i+1) for i in range(200))
sage: G.add_edge(200,0)
sage: G.add_edge(1,0)
sage: e_min, e_max = spectral_radius(G, 0.00001)
sage: p = G.adjacency_matrix(sparse=True).charpoly()
sage: p
x^201 - x^199 - 1
sage: r = p.roots(AA, multiplicities=False)[0]
sage: e_min < r < e_max
True
```

A much larger example:

```
sage: G = DiGraph(100000)
sage: r = list(range(100000))
sage: while not G.is_strongly_connected():
....:     shuffle(r)
....:     G.add_edges(enumerate(r), loops=False)
sage: spectral_radius(G, 1e-10)  # random
(1.9997956006500042, 1.9998043797692782)
```

The algorithm takes care of multiple edges:

```
sage: G = DiGraph(2,loops=True,multiedges=True)
sage: G.add_edges([(0,0),(0,0),(0,1),(1,0)])
sage: spectral_radius(G, 1e-14)  # abs tol 1e-14
(2.414213562373094, 2.414213562373095)
sage: max(G.adjacency_matrix().eigenvalues(AA))
2.414213562373095?
```

Some bipartite graphs:

```
sage: G = Graph([(0,1),(0,3),(2,3)])
sage: G.spectral_radius()  # abs tol 1e-10
(1.6180339887253428, 1.6180339887592732)

sage: G = DiGraph([(0,1),(0,3),(2,3),(3,0),(1,0),(1,2)])
sage: G.spectral_radius() # abs tol 1e-10
(1.5537739740270458, 1.553773974033029)

sage: G = graphs.CompleteBipartiteGraph(1,3)
sage: G.spectral_radius()  # abs tol 1e-10
(1.7320508075688772, 1.7320508075688774)
```

sage.graphs.base.static_sparse_graph.**strongly_connected_components_digraph**(*G*)
Return the digraph of the strongly connected components (SCCs).

This routine is used to test `strongly_connected_components_digraph_C`, but it is not used by the Sage digraph. It outputs a pair `[g_scc,scc]`, where `g_scc` is the SCC digraph of g, `scc` is a dictionary associating to each vertex `v` the number of the SCC of `v`, as it appears in `g_scc`.

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import strongly_connected_
↪components_digraph
sage: strongly_connected_components_digraph(digraphs.Path(3))
(Digraph on 3 vertices, {0: 2, 1: 1, 2: 0})
sage: strongly_connected_components_digraph(DiGraph(4))
(Digraph on 4 vertices, {0: 0, 1: 1, 2: 2, 3: 3})
```

sage.graphs.base.static_sparse_graph.**tarjan_strongly_connected_components**(*G*)
Return the lists of vertices in each strongly connected components (SCCs).

This method implements the Tarjan algorithm to compute the strongly connected components of the digraph. It returns a list of lists of vertices, each list of vertices representing a strongly connected component.

The basic idea of the algorithm is this: a depth-first search (DFS) begins from an arbitrary start node (and subsequent DFSes are conducted on any nodes that have not yet been found). As usual with DFSes, the search visits every node of the graph exactly once, declining to revisit any node that has already been explored. Thus, the collection of search trees is a spanning forest of the graph. The strongly connected components correspond to the subtrees of this spanning forest that have no edge directed outside the subtree.

To recover these components, during the DFS, we keep the index of a node, that is, the position in the DFS tree, and the lowlink: as soon as the subtree rooted at v has been fully explored, the lowlink of v is the smallest index reachable from v passing from descendants of v. If the subtree rooted at v has been fully explored, and the index of v equals the lowlink of v, that whole subtree is a new SCC.

For more information, see the Wikipedia article Tarjan's_strongly_connected_components_algorithm.

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import tarjan_strongly_connected_
↪components
sage: tarjan_strongly_connected_components(digraphs.Path(3))
[[2], [1], [0]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.strongly_connected_components()
[[3], [2], [1], [0], [6], [5], [4]]
```

(continues on next page)

(continued from previous page)

```
sage: D.add_edge([2,0])
sage: D.strongly_connected_components()
[[3], [0, 1, 2], [6], [5], [4]]
sage: D = DiGraph([('a','b'), ('b','c'), ('c', 'd'), ('d', 'b'), ('c', 'e')])
sage: [sorted(scc) for scc in D.strongly_connected_components()]
[['e'], ['b', 'c', 'd'], ['a']]
```

sage.graphs.base.static_sparse_graph.**triangles_count**(*G*)

Return the number of triangles containing v, for every v.

INPUT:

- *G*– a graph

EXAMPLES:

```
sage: from sage.graphs.base.static_sparse_graph import triangles_count
sage: triangles_count(graphs.PetersenGraph())
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0}
sage: sum(triangles_count(graphs.CompleteGraph(15)).values()) == 3*binomial(15,3)
True
```

3.7 Static sparse graph backend

This module implement a immutable sparse graph backend using the data structure from *sage.graphs.base.static_sparse_graph*. It supports both directed and undirected graphs, as well as vertex/edge labels, loops and multiple edges. As it uses a very compact C structure it should be very small in memory.

As it is a sparse data structure, you can expect it to be very efficient when you need to list the graph's edge, or those incident to a vertex, but an adjacency test can be much longer than in a dense data structure (i.e. like in *sage.graphs.base.static_dense_graph*)

For an overview of graph data structures in sage, see *overview*.

3.7.1 Two classes

This module implements two classes

- *StaticSparseCGraph* extends *CGraph* and is a Cython class that manages the definition/deallocation of the `short_digraph` structure. It does not know anything about labels on vertices.
- *StaticSparseBackend* extends *CGraphBackend* and is a Python class that does know about vertex labels and contains an instance of *StaticSparseCGraph* as an internal variable. The input/output of its methods are labeled vertices, which it translates to integer id before forwarding them to the *StaticSparseCGraph* instance.

3.7.2 Classes and methods

class sage.graphs.base.static_sparse_backend.**StaticSparseBackend**

Bases: *sage.graphs.base.c_graph.CGraphBackend*

A graph *backend* for static sparse graphs.

EXAMPLES:

```
sage: D = sage.graphs.base.sparse_graph.SparseGraphBackend(9)
sage: D.add_edge(0, 1, None, False)
sage: list(D.iterator_edges(range(9), True))
[(0, 1, None)]
```

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: list(g.iterator_edges([0], 1))
[(0, 1, None), (0, 4, None), (0, 5, None)]
```

```
sage: g = DiGraph(digraphs.DeBruijn(4, 3), data_structure="static_sparse")
sage: gi = DiGraph(g, data_structure="static_sparse")
sage: gi.edges()[0]
('000', '000', '0')
sage: sorted(gi.edges_incident('111'))
[('111', '110', '0'),
('111', '111', '1'),
('111', '112', '2'),
('111', '113', '3')]

sage: set(g.edges()) == set(gi.edges())
True
```

```
sage: g = graphs.PetersenGraph()
sage: gi = Graph(g, data_structure="static_sparse")
sage: g == gi
True
sage: set(g.edges()) == set(gi.edges())
True
```

```
sage: gi = Graph({ 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2}}, data_
↪structure="static_sparse")
sage: (0, 4, 2) in gi.edges()
True
sage: gi.has_edge(0, 4)
True
```

```
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: GI = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}}, data_structure="static_sparse")
sage: G == GI
True
```

```
sage: G = graphs.OddGraph(4)
sage: d = G.diameter()
sage: H = G.distance_graph(list(range(d + 1)))
sage: HI = Graph(H, data_structure="static_sparse")
sage: HI.size() == len(HI.edges())
True
```

```
sage: g = Graph({1: {1: [1, 2, 3]}}, data_structure="static_sparse")
sage: g.size()
3
sage: g.order()
```

(continues on next page)

(continued from previous page)

```
1
sage: g.vertices()
[1]
sage: g.edges()
[(1, 1, 1), (1, 1, 2), (1, 1, 3)]
```

trac ticket #15810 is fixed:

```
sage: DiGraph({1: {2: ['a', 'b'], 3: ['c']}, 2: {3: ['d']}}, immutable=True).is_
↪directed_acyclic()
True
```

add_edge(*u*, *v*, *l*, *directed*)
Set edge label. No way.

add_edges(*edges*, *directed*)
Set edge label. No way.

add_vertex(*v*)
Addition of vertices is not available on an immutable graph.

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.add_vertex(1)
Traceback (most recent call last):
...
ValueError: graph is immutable; please change a copy instead (use function␣
↪copy())
sage: g.add_vertices([1,2,3])
Traceback (most recent call last):
...
ValueError: graph is immutable; please change a copy instead (use function␣
↪copy())
```

add_vertices(*vertices*)
Set edge label. No way.

allows_loops(*value=None*)
Return whether the graph allows loops

INPUT:

- `value` – only useful for compatibility with other graph backends, where this method can be used to define this boolean. This method raises an exception if `value` is not equal to `None`.

degree(*v*, *directed*)
Return the degree of a vertex

INPUT:

- `v` – a vertex
- `directed` – boolean; whether to take into account the orientation of this graph in counting the degree of `v`

EXAMPLES:

```
sage: g = Graph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.degree(0)
3
```

trac ticket #17225 about the degree of a vertex with a loop:

```
sage: Graph({0: [0]}, immutable=True).degree(0)
2
sage: Graph({0: [0], 1: [0, 1, 1, 1]}, immutable=True).degree(1)
7
```

del_edge(*u*, *v*, *l*, *directed*)
Set edge label. No way.

del_vertex(*v*)
Removal of vertices is not available on an immutable graph.

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.delete_vertex(1)
Traceback (most recent call last):
...
ValueError: graph is immutable; please change a copy instead (use function␣
↪copy())
sage: g.delete_vertices([1,2,3])
Traceback (most recent call last):
...
ValueError: graph is immutable; please change a copy instead (use function␣
↪copy())
```

get_edge_label(*u*, *v*)
Return the edge label for `(u, v)`.

INPUT:

- `u, v` – two vertices

has_edge(*u*, *v*, *l*)
Return whether this graph has edge `(u, v)` with label `l`.

If `l` is `None`, return whether this graph has an edge `(u, v)` with any label.

INPUT:

- `u, v` – two vertices
- `l` – a label

has_vertex(*v*)
Test if the vertex belongs to the graph

INPUT:

- `v` – a vertex (or not?)

in_degree(*v*)
Return the in-degree of a vertex

INPUT:

- `v` – a vertex

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.in_degree(0)
3
```

iterator_edges(*vertices*, *labels*)
Return an iterator over the graph's edges.

INPUT:

- vertices – list; only returns the edges incident to at least one vertex of vertices
- labels – boolean; whether to return edge labels too

iterator_in_edges(*vertices*, *labels*)
Iterate over the incoming edges incident to a sequence of vertices.

INPUT:

- vertices – a list of vertices
- labels – whether to return labels too

iterator_in_nbrs(*v*)
Return an iterator over the in-neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.neighbors_in(0)
[1, 4, 5]
```

iterator_nbrs(*v*)
Return an iterator over the neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLES:

```
sage: g = Graph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.neighbors(0)
[1, 4, 5]
```

iterator_out_edges(*vertices*, *labels*)
Iterate over the outbound edges incident to a sequence of vertices.

INPUT:

- vertices – a list of vertices
- labels – whether to return labels too

iterator_out_nbrs(*v*)
Return an iterator over the out-neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.neighbors_out(0)
[1, 4, 5]
```

iterator_verts(*vertices*)
Return an iterator over the vertices

INPUT:

- `vertices` – a list of objects; the method will only return the elements of the graph which are contained in `vertices`. It's not very efficient. If `vertices` is equal to `None`, all the vertices are returned.

multiple_edges(*value=None*)
Return whether the graph allows multiple edges

INPUT:

- `value` – only useful for compatibility with other graph backends, where this method can be used to define this boolean. This method raises an exception if `value` is not equal to `None`.

num_edges(*directed*)
Return the number of edges

INPUT:

- `directed` – boolean; whether to consider the graph as directed or not.

num_verts()
Return the number of vertices

out_degree(*v*)
Return the out-degree of a vertex

INPUT:

- `v` – a vertex

EXAMPLES:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.out_degree(0)
3
```

relabel(*perm*, *directed*)
Relabel the graphs' vertices. No way.

set_edge_label(*u*, *v*, *l*, *directed*)
Set edge label. No way.

class sage.graphs.base.static_sparse_backend.**StaticSparseCGraph**
Bases: *sage.graphs.base.c_graph.CGraph*

CGraph class based on the sparse graph data structure *static sparse graphs*.

add_vertex(*k*)
Add a vertex to the graph. No way.

del_vertex(*k*)
Remove a vertex from the graph. No way.

has_arc(*u*, *v*)
Test if uv is an edge of the graph

INPUT:

- u, v – integers

has_vertex(*v*)
Test if a vertex belongs to the graph

INPUT:

- n – an integer

in_degree(*u*)
Return the in-degree of a vertex

INPUT:

- u – a vertex

in_neighbors(*u*)
Return the in-neighbors of a vertex

INPUT:

- u – a vertex

out_degree(*u*)
Return the out-degree of a vertex

INPUT:

- u – a vertex

out_neighbors(*u*)
List the out-neighbors of a vertex

INPUT:

- u – a vertex

verts()
Returns the list of vertices

3.8 Backends for Sage (di)graphs.

This module implements *GenericGraphBackend* (the base class for backends).

Any graph backend must redefine the following methods (for which *GenericGraphBackend* raises a NotImplementedError)

add_edge()	Add an edge (u, v) to `self`, with label l.
add_edges()	Add a sequence of edges to `self`.
add_vertex()	Add a labelled vertex to `self`.
add_vertices()	Add labelled vertices to `self`.
degree()	Return the total number of vertices incident to v.
in_degree()	Return the in-degree of v
out_degree()	Return the out-degree of v
del_edge()	Delete the edge (u, v) with label l.
del_vertex()	Delete a labelled vertex in `self`.
del_vertices()	Delete labelled vertices in `self`.
get_edge_label()	Return the edge label of (u, v).
has_edge()	True if `self` has an edge (u, v) with label l.
has_vertex()	True if `self` has a vertex with label v.
iterator_edges()	Iterate over the edges incident to a sequence of vertices.
iterator_in_edges()	Iterate over the incoming edges incident to a sequence of vertices.
iterator_out_edges()	Iterate over the outbound edges incident to a sequence of vertices.
iterator_nbrs()	Iterate over the vertices adjacent to v.
iterator_in_nbrs()	Iterate over the in-neighbors of vertex v.
iterator_out_nbrs()	Iterate over the out-neighbors of vertex v.
iterator_verts()	Iterate over the vertices v with labels in verts.
loops()	Get/set whether or not `self` allows loops.
multiple_edges()	Get/set whether or not `self` allows multiple edges.
name()	Get/set name of `self`.
num_edges()	The number of edges in `self`
num_verts()	The number of vertices in `self`
relabel()	Relabel the vertices of `self` by a permutation.
set_edge_label()	Label the edge (u, v) by l.

For an overview of graph data structures in sage, see *overview*.

3.8.1 Classes and methods

class sage.graphs.base.graph_backends.**GenericGraphBackend**

Bases: sage.structure.sage_object.SageObject

A generic wrapper for the backend of a graph.

Various graph classes use extensions of this class. Note, this graph has a number of placeholder functions, so the doctests are rather silly.

add_edge(*u*, *v*, *l*, *directed*)

Add an edge (u, v) to `self`, with label l.

If `directed` is `True`, this is interpreted as an arc from u to v.

INPUT:

- `u, v` – vertices
- `l` – edge label
- `directed` – boolean

add_edges(*edges*, *directed*)

Add a sequence of edges to `self`.

If `directed` is `True`, these are interpreted as arcs.

INPUT:

- `edges` – list/iterator of edges to be added
- `directed` – boolean

add_vertex(*name*)
Add a labelled vertex to `self`.

INPUT:

- `name` – vertex label

OUTPUT:

If `name=None`, the new vertex name is returned, `None` otherwise.

add_vertices(*vertices*)
Add labelled vertices to `self`.

INPUT:

- `vertices` – iterator of vertex labels; a new label is created, used and returned in the output list for all `None` values in `vertices`

OUTPUT:

Generated names of new vertices if there is at least one `None` value present in `vertices`. `None` otherwise.

EXAMPLES:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.add_vertices([1,2,3])
Traceback (most recent call last):
...
NotImplementedError
```

degree(*v*, *directed*)
Return the total number of vertices incident to v.

INPUT:

- `v` – a vertex label
- `directed` – boolean

OUTPUT:

degree of v

del_edge(*u*, *v*, *l*, *directed*)
Delete the edge (u, v) with label l.

INPUT:

- `u,v` – vertices
- `l` – edge label
- `directed` – boolean

del_vertex(*v*)
Delete a labelled vertex in `self`.

INPUT:

- v – vertex label

del_vertices(*vertices*)
Delete labelled vertices in self.

INPUT:

- vertices – iterator of vertex labels

get_edge_label(*u*, *v*)
Return the edge label of (u, v).

INPUT:

- u, v – vertex labels

OUTPUT:

label of (u, v)

has_edge(*u*, *v*, *l*)
Check whether self has an edge (u, v) with label l.

INPUT:

- u, v – vertex labels
- l – label

OUTPUT:

boolean

has_vertex(*v*)
Check whether self has a vertex with label v.

INPUT:

- v – vertex label

OUTPUT: boolean

in_degree(*v*)
Return the in-degree of v

INPUT:

- v – a vertex label

iterator_edges(*vertices*, *labels*)
Iterate over the edges incident to a sequence of vertices.

Edges are assumed to be undirected.

This method returns an iterator over the edges (u, v) such that either u or v is in vertices and the edge (u, v) is in self.

INPUT:

- vertices – a list of vertex labels
- labels – boolean

OUTPUT:

a generator which yields edges, with or without labels depending on the labels parameter.

iterator_in_edges(*vertices*, *labels*)
Iterate over the incoming edges incident to a sequence of vertices.

This method returns an iterator over the edges (u, v) such that v is in `vertices` and the edge (u, v) is in `self`.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean

OUTPUT: a generator which yields edges, with or without labels depending on the labels parameter.

iterator_in_nbrs(v)
Iterate over the in-neighbors of vertex v.

This method returns an iterator over the vertices u such that the edge (u, v) is in `self` (that is, predecessors of v).

INPUT:

- `v` – vertex label

OUTPUT:

a generator which yields vertex labels

iterator_nbrs(v)
Iterate over the vertices adjacent to v.

This method returns an iterator over the vertices u such that either the edge (u, v) or the edge (v, u) is in `self` (that is, neighbors of v).

INPUT:

- `v` – vertex label

OUTPUT:

a generator which yields vertex labels

iterator_out_edges(*vertices*, *labels*)
Iterate over the outbound edges incident to a sequence of vertices.

This method returns an iterator over the edges (v, u) such that v is in `vertices` and the edge (v, u) is in `self`.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean

OUTPUT:

a generator which yields edges, with or without labels depending on the labels parameter.

iterator_out_nbrs(v)
Iterate over the out-neighbors of v.

This method returns an iterator over the vertices u such that the edge (v, u) is in `self` (that is, successors of v).

INPUT:

- `v` – vertex label

OUTPUT:

a generator which yields vertex labels

iterator_verts(*verts*)
Iterate over the vertices v with labels in `verts`.

INPUT:

- `verts` – vertex labels

OUTPUT:

a generator which yields vertices

loops(*new=None*)
Get/set whether or not self allows loops.

INPUT:

- `new` – can be a boolean (in which case it sets the value) or `None`, in which case the current value is returned. It is set to `None` by default.

multiple_edges(*new=None*)
Get/set whether or not self allows multiple edges.

INPUT:

- `new` – can be a boolean (in which case it sets the value) or `None`, in which case the current value is returned. It is set to `None` by default.

name(*new=None*)
Get/set name of self.

INPUT:

- `new` – can be a string (in which case it sets the value) or `None`, in which case the current value is returned. It is set to `None` by default.

num_edges(*directed*)
Return the number of edges in `self`

INPUT:

- `directed` – boolean

num_verts()
Return the number of vertices in `self`

out_degree(*v*)
Return the out-degree of v

INPUT:

- `v` – a vertex label

relabel(*perm*, *directed*)
Relabel the vertices of `self` by a permutation.

INPUT:

- `perm` – permutation
- `directed` – boolean

set_edge_label(*u*, *v*, *l*, *directed*)
Label the edge (u, v) by l.

INPUT:

- `u,v` – vertices
- `l` – edge label
- `directed` – boolean

`sage.graphs.base.graph_backends.`**unpickle_graph_backend**(*directed*, *vertices*, *edges*, *kwds*)
Return a backend from its pickled data

This methods is defined because Python's pickling mechanism can only build objects from a pair `(f,args)` by running `f(*args)`. In particular, there is apparently no way to define a `**kwargs` (i.e. define the value of keyword arguments of `f`), which means that one must know the order of all arguments of `f` (here, `f` is *Graph* or *DiGraph*).

As a consequence, this means that the order cannot change in the future, which is something we cannot swear.

INPUT:

- `directed` – boolean
- `vertices` – list of vertices
- `edges` – list of edges
- `kwds` – any dictionary whose keywords will be forwarded to the graph constructor

This function builds a *Graph* or *DiGraph* from its data, and returns the `_backend` attribute of this object.

EXAMPLES:

```
sage: from sage.graphs.base.graph_backends import unpickle_graph_backend
sage: b = unpickle_graph_backend(0, [0, 1, 2, 3], [(0, 3, 'label'), (0, 0, 1)], {
↪'loops': True})
sage: b
<sage.graphs.base.sparse_graph.SparseGraphBackend object at ...>
sage: list(b.iterator_edges(range(4), True))
[(0, 0, 1), (0, 3, 'label')]
```

3.9 Interface to run Boost algorithms

Wrapper for a Boost graph. The Boost graphs are Cython C++ variables, and they cannot be converted to Python objects: as a consequence, only functions defined with cdef are able to create, read, modify, and delete these graphs.

A very important feature of Boost graph library is that all object are generic: for instance, adjacency lists can be stored using different data structures, and (most of) the functions work with all implementations provided. This feature is implemented in our interface using fused types: however, Cython's support for fused types is still experimental, and some features are missing. For instance, there cannot be nested generic function calls, and no variable can have a generic type, apart from the arguments of a generic function.

All the input functions use pointers, because otherwise we might have problems with `delete()`.

Basic Boost Graph operations:

clustering_coeff()	Return the clustering coefficient of all vertices in the graph.
edge_connectivity()	Return the edge connectivity of the graph.
dominator_tree()	Return a dominator tree of the graph.
bandwidth_heuristics()	Use heuristics to approximate the bandwidth of the graph.
min_spanning_tree()	Compute a minimum spanning tree of a (weighted) graph.
shortest_paths()	Use Dijkstra or Bellman-Ford algorithm to compute the single-source shortest paths.
johnson_shortest_paths()	Use Johnson algorithm to compute the all-pairs shortest paths.
floyd_warshall_shortest_paths()	Use Floyd-Warshall algorithm to compute the all-pairs shortest paths.
johnson_closeness_centrality()	Use Johnson algorithm to compute the closeness centrality of all vertices.
blocks_and_cut_vertices()	Use Tarjan's algorithm to compute the blocks and cut vertices of the graph.
min_cycle_basis()	Return a minimum weight cycle basis of the input graph.

3.9.1 Functions

sage.graphs.base.boost_graph.**bandwidth_heuristics**(*g*, *algorithm='cuthill_mckee'*)

Use Boost heuristics to approximate the bandwidth of the input graph.

The bandwidth $bw(M)$ of a matrix M is the smallest integer k such that all non-zero entries of M are at distance k from the diagonal. The bandwidth $bw(g)$ of an undirected graph g is the minimum bandwidth of the adjacency matrix of g, over all possible relabellings of its vertices (for more information, see the *bandwidth* module).

Unfortunately, exactly computing the bandwidth is NP-hard (and an exponential algorithm is implemented in Sagemath in routine *bandwidth()*). Here, we implement two heuristics to find good orderings: Cuthill-McKee, and King.

This function works only in undirected graphs, and its running time is $O(md_{max}\log d_{max})$ for the Cuthill-McKee ordering, and $O(md_{max}^2\log d_{max})$ for the King ordering, where m is the number of edges, and d_{max} is the maximum degree in the graph.

INPUT:

- `g` – the input Sage graph
- `algorithm` – string (default: `'cuthill_mckee'`); the heuristic used to compute the ordering among `'cuthill_mckee'` and `'king'`

OUTPUT:

A pair `[bandwidth, ordering]`, where `ordering` is the ordering of vertices, `bandwidth` is the bandwidth of that specific ordering (which is not necessarily the bandwidth of the graph, because this is a heuristic).

EXAMPLES:

```
sage: from sage.graphs.base.boost_graph import bandwidth_heuristics
sage: bandwidth_heuristics(graphs.PathGraph(10))
(1, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
sage: bandwidth_heuristics(graphs.GridGraph([3,3]))  # py2
(3, [(2, 2), (2, 1), (1, 2), (2, 0), (1, 1), (0, 2), (1, 0), (0, 1), (0, 0)])
sage: bandwidth_heuristics(graphs.GridGraph([3,3]), algorithm='king')  # py2
(3, [(2, 2), (2, 1), (1, 2), (2, 0), (1, 1), (0, 2), (1, 0), (0, 1), (0, 0)])
sage: bandwidth_heuristics(graphs.GridGraph([3,3]))  # py3
(3, [(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (2, 1), (1, 2), (2, 2)])
sage: bandwidth_heuristics(graphs.GridGraph([3,3]), algorithm='king')  # py3
(3, [(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (2, 1), (1, 2), (2, 2)])
```

sage.graphs.base.boost_graph.**blocks_and_cut_vertices**(*g*)
Compute the blocks and cut vertices of the graph.

This method uses the implementation of Tarjan's algorithm available in the Boost library .

INPUT:

- g – the input Sage graph

OUTPUT:

A 2-dimensional vector with m+1 rows (m is the number of biconnected components), where each of the first m rows correspond to vertices in a block, and the last row is the list of cut vertices.

See also:

- *sage.graphs.generic_graph.GenericGraph.blocks_and_cut_vertices()*

EXAMPLES:

```
sage: from sage.graphs.base.boost_graph import blocks_and_cut_vertices
sage: g = graphs.KrackhardtKiteGraph()
sage: blocks_and_cut_vertices(g)
([[8, 9], [7, 8], [0, 1, 2, 3, 5, 4, 6, 7]], [8, 7])

sage: G = Graph([(0,1,{'name':'a','weight':1}), (0,2,{'name':'b','weight':3}), (1,
↪2,{'name':'b','weight':1})])
sage: blocks_and_cut_vertices(G)
([[0, 1, 2]], [])
```

sage.graphs.base.boost_graph.**clustering_coeff**(*g*, *vertices=None*)
Compute the clustering coefficient of the input graph, using Boost.

See also:

sage.graphs.generic_graph.GenericGraph.clustering_coeff()

INPUT:

- g – the input Sage Graph
- vertices – list (default: None); the list of vertices to analyze (if None, compute the clustering coefficient of all vertices)

OUTPUT: a pair (average_clustering_coefficient, clust_of_v), where average_clustering_coefficient is the average clustering of the vertices in variable vertices, clust_of_v is a dictionary that associates to each vertex its clustering coefficient. If vertices is None, all vertices are considered.

EXAMPLES:

Computing the clustering coefficient of a clique:

```
sage: from sage.graphs.base.boost_graph import clustering_coeff
sage: g = graphs.CompleteGraph(5)
sage: clustering_coeff(g)
(1.0, {0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0})
sage: clustering_coeff(g, vertices = [0,1,2])
(1.0, {0: 1.0, 1: 1.0, 2: 1.0})
```

Of a non-clique graph with triangles:

```
sage: g = graphs.IcosahedralGraph()
sage: clustering_coeff(g, vertices=[1,2,3])
(0.5, {1: 0.5, 2: 0.5, 3: 0.5})
```

With labels:

```
sage: g.relabel(list("abcdefghiklm"))
sage: clustering_coeff(g, vertices="abde")
(0.5, {'a': 0.5, 'b': 0.5, 'd': 0.5, 'e': 0.5})
```

sage.graphs.base.boost_graph.**dominator_tree**(*g*, *root*, *return_dict=False*, *reverse=False*)

Use Boost to compute the dominator tree of `g`, rooted at `root`.

A node d dominates a node n if every path from the entry node `root` to n must go through d. The immediate dominator of a node n is the unique node that strictly dominates n but does not dominate any other node that dominates n. A dominator tree is a tree where each node's children are those nodes it immediately dominates. For more information, see the Wikipedia article Dominator_(graph_theory).

If the graph is connected and undirected, the parent of a vertex v is:

- the root if v is in the same biconnected component as the root;
- the first cut vertex in a path from v to the root, otherwise.

If the graph is not connected, the dominator tree of the whole graph is equal to the dominator tree of the connected component of the root.

If the graph is directed, computing a dominator tree is more complicated, and it needs time $O(m \log m)$, where m is the number of edges. The implementation provided by Boost is the most general one, so it needs time $O(m \log m)$ even for undirected graphs.

INPUT:

- `g` – the input Sage (Di)Graph
- `root` – the root of the dominator tree
- `return_dict` – boolean (default: `False`); if `True`, the function returns a dictionary associating to each vertex its parent in the dominator tree. If `False` (default), it returns the whole tree, as a `Graph` or a `DiGraph`.
- `reverse` – boolean (default: `False`); when set to `True`, computes the dominator tree in the reverse graph

OUTPUT:

The dominator tree, as a graph or as a dictionary, depending on the value of `return_dict`. If the output is a dictionary, it will contain `None` in correspondence of `root` and of vertices that are not reachable from `root`. If the output is a graph, it will not contain vertices that are not reachable from `root`.

EXAMPLES:

An undirected grid is biconnected, and its dominator tree is a star (everyone's parent is the root):

```
sage: g = graphs.GridGraph([2,2]).dominator_tree((0,0))
sage: g.to_dictionary()
{(0, 0): [(0, 1), (1, 0), (1, 1)], (0, 1): [(0, 0)], (1, 0): [(0, 0)], (1, 1):␣
↪[(0, 0)]}
```

If the graph is made by two 3-cycles C_1, C_2 connected by an edge (v, w), with $v \in C_1$, $w \in C_2$, the cut vertices are v and w, the biconnected components are C_1, C_2, and the edge (v, w). If the root is in C_1, the parent of each vertex in C_1 is the root, the parent of w is v, and the parent of each vertex in C_2 is w:

```
sage: G = 2 * graphs.CycleGraph(3)
sage: v = 0
sage: w = 3
sage: G.add_edge(v,w)
sage: G.dominator_tree(1, return_dict=True)
{0: 1, 1: None, 2: 1, 3: 0, 4: 3, 5: 3}
```

An example with a directed graph:

```
sage: g = digraphs.Circuit(10).dominator_tree(5)
sage: g.to_dictionary()
{0: [1], 1: [2], 2: [3], 3: [4], 4: [], 5: [6], 6: [7], 7: [8], 8: [9], 9: [0]}
sage: g = digraphs.Circuit(10).dominator_tree(5, reverse=True)
sage: g.to_dictionary()
{0: [9], 1: [0], 2: [1], 3: [2], 4: [3], 5: [4], 6: [], 7: [6], 8: [7], 9: [8]}
```

If the output is a dictionary:

```
sage: graphs.GridGraph([2,2]).dominator_tree((0,0), return_dict=True)
{(0, 0): None, (0, 1): (0, 0), (1, 0): (0, 0), (1, 1): (0, 0)}
```

sage.graphs.base.boost_graph.**edge_connectivity**(*g*)

Compute the edge connectivity of the input graph, using Boost.

OUTPUT: a pair `(ec, edges)`, where `ec` is the edge connectivity, `edges` is the list of edges in a minimum cut.

See also:

sage.graphs.generic_graph.GenericGraph.edge_connectivity()

EXAMPLES:

Computing the edge connectivity of a clique:

```
sage: from sage.graphs.base.boost_graph import edge_connectivity
sage: g = graphs.CompleteGraph(5)
sage: edge_connectivity(g)
(4, [(0, 1), (0, 2), (0, 3), (0, 4)])
```

Vertex-labeled graphs:

```
sage: from sage.graphs.base.boost_graph import edge_connectivity
sage: g = graphs.GridGraph([2,2])
sage: edge_connectivity(g)  # py2
(2, [((0, 1), (1, 1)), ((0, 1), (0, 0))])
sage: edge_connectivity(g)  # py3
(2, [((0, 0), (0, 1)), ((0, 0), (1, 0))])
```

sage.graphs.base.boost_graph.**floyd_warshall_shortest_paths**(*g*, *weight_function=None*, *distances=True*, *predecessors=False*)

Use Floyd-Warshall algorithm to solve the all-pairs-shortest-paths.

This routine outputs the distance between each pair of vertices and the predecessors matrix (depending on the values of boolean `distances` and `predecessors`) using a dictionary of dictionaries. This method should be preferred only if the graph is dense. If the graph is sparse the much faster johnson_shortest_paths should be used.

The time-complexity is $O(n^3 + nm)$, where n is the number of nodes and m the number of edges. The factor nm in the complexity is added only when `predecessors` is set to `True`.

INPUT:

- `g` – the input Sage graph
- `weight_function` – function (default: `None`); a function that associates a weight to each edge. If `None` (default), the weights of `g` are used, if available, otherwise all edges have weight 1.
- `distances` – boolean (default: `True`); whether to return the dictionary of shortest distances
- `predecessors` – boolean (default: `False`); whether to return the predecessors matrix

OUTPUT:

Depending on the input, this function return the dictionary of predecessors, the dictionary of distances, or a pair of dictionaries `(distances, predecessors)` where `distance[u][v]` denotes the distance of a shortest path from u to v and `predecessors[u][v]` indicates the predecessor of w on a shortest path from u to v.

EXAMPLES:

Undirected graphs:

```
sage: from sage.graphs.base.boost_graph import floyd_warshall_shortest_paths
sage: g = Graph([(0,1,1),(1,2,2),(1,3,4),(2,3,1)], weighted=True)
sage: floyd_warshall_shortest_paths(g)
{0: {0: 0, 1: 1, 2: 3, 3: 4},
 1: {0: 1, 1: 0, 2: 2, 3: 3},
 2: {0: 3, 1: 2, 2: 0, 3: 1},
 3: {0: 4, 1: 3, 2: 1, 3: 0}}
sage: expected = {0: {0: None, 1: 0, 2: 1, 3: 2},
....:             1: {0: 1, 1: None, 2: 1, 3: 2},
....:             2: {0: 1, 1: 2, 2: None, 3: 2},
....:             3: {0: 1, 1: 2, 2: 3, 3: None}}
sage: floyd_warshall_shortest_paths(g, distances=False, predecessors=True) ==␣
↪expected
True
```

Directed graphs:

```
sage: g = DiGraph([(0,1,1),(1,2,-2),(1,3,4),(2,3,1)], weighted=True)
sage: floyd_warshall_shortest_paths(g)
{0: {0: 0, 1: 1, 2: -1, 3: 0},
 1: {1: 0, 2: -2, 3: -1},
 2: {2: 0, 3: 1},
 3: {3: 0}}
sage: g = DiGraph([(1,2,3),(2,3,2),(1,4,1),(4,2,1)], weighted=True)
sage: floyd_warshall_shortest_paths(g, distances=False, predecessors=True)
{1: {1: None, 2: 4, 3: 2, 4: 1},
 2: {2: None, 3: 2},
 3: {3: None},
 4: {2: 4, 3: 2, 4: None}}
```

sage.graphs.base.boost_graph.**johnson_closeness_centrality**(*g*, *weight_function=None*)

Use Johnson algorithm to compute the closeness centrality of all vertices.

This routine is preferrable to *johnson_shortest_paths()* because it does not create a doubly indexed dictionary of distances, saving memory.

The time-complexity is $O(mn \log n)$, where n is the number of nodes and m is the number of edges.

INPUT:

- `g` – the input Sage graph
- `weight_function` – function (default: `None`); a function that associates a weight to each edge. If `None` (default), the weights of `g` are used, if available, otherwise all edges have weight 1.

OUTPUT:

A dictionary associating each vertex `v` to its closeness centrality.

EXAMPLES:

Undirected graphs:

```
sage: from sage.graphs.base.boost_graph import johnson_closeness_centrality
sage: g = Graph([(0,1,1),(1,2,2),(1,3,4),(2,3,1)], weighted=True)
sage: johnson_closeness_centrality(g)
{0: 0.375, 1: 0.5, 2: 0.5, 3: 0.375}
```

Directed graphs:

```
sage: from sage.graphs.base.boost_graph import johnson_closeness_centrality
sage: g = DiGraph([(0,1,1),(1,2,-2),(1,3,4),(2,3,1)], weighted=True)
sage: johnson_closeness_centrality(g)
{0: inf, 1: -0.4444444444444444, 2: 0.3333333333333333}
```

sage.graphs.base.boost_graph.**johnson_shortest_paths**(*g*, *weight_function=None*, *distances=True*, *predecessors=False*)

Use Johnson algorithm to solve the all-pairs-shortest-paths.

This routine outputs the distance between each pair of vertices and the predecessors matrix (depending on the values of boolean `distances` and `predecessors`) using a dictionary of dictionaries. It works on all kinds of graphs, but it is designed specifically for graphs with negative weights (otherwise there are more efficient algorithms, like Dijkstra).

The time-complexity is $O(mn \log n)$, where n is the number of nodes and m is the number of edges.

INPUT:

- `g` – the input Sage graph
- `weight_function` – function (default: `None`); a function that associates a weight to each edge. If `None` (default), the weights of `g` are used, if available, otherwise all edges have weight 1.
- `distances` – boolean (default: `True`); whether to return the dictionary of shortest distances
- `predecessors` – boolean (default: `False`); whether to return the predecessors matrix

OUTPUT:

Depending on the input, this function return the dictionary of predecessors, the dictionary of distances, or a pair of dictionaries `(distances, predecessors)` where `distance[u][v]` denotes the distance of a shortest path from u to v and `predecessors[u][v]` indicates the predecessor of w on a shortest path from u to v.

EXAMPLES:

Undirected graphs:

```
sage: from sage.graphs.base.boost_graph import johnson_shortest_paths
sage: g = Graph([(0,1,1),(1,2,2),(1,3,4),(2,3,1)], weighted=True)
sage: johnson_shortest_paths(g)
{0: {0: 0, 1: 1, 2: 3, 3: 4},
 1: {0: 1, 1: 0, 2: 2, 3: 3},
 2: {0: 3, 1: 2, 2: 0, 3: 1},
 3: {0: 4, 1: 3, 2: 1, 3: 0}}
sage: expected = {0: {0: None, 1: 0, 2: 1, 3: 2},
....:             1: {0: 1, 1: None, 2: 1, 3: 2},
....:             2: {0: 1, 1: 2, 2: None, 3: 2},
....:             3: {0: 1, 1: 2, 2: 3, 3: None}}
sage: johnson_shortest_paths(g, distances=False, predecessors=True) == expected
True
```

Directed graphs:

```
sage: g = DiGraph([(0,1,1),(1,2,-2),(1,3,4),(2,3,1)], weighted=True)
sage: johnson_shortest_paths(g)
{0: {0: 0, 1: 1, 2: -1, 3: 0},
 1: {1: 0, 2: -2, 3: -1},
 2: {2: 0, 3: 1},
 3: {3: 0}}
sage: g = DiGraph([(1,2,3),(2,3,2),(1,4,1),(4,2,1)], weighted=True)
sage: johnson_shortest_paths(g, distances=False, predecessors=True)
{1: {1: None, 2: 4, 3: 2, 4: 1},
 2: {2: None, 3: 2},
 3: {3: None},
 4: {2: 4, 3: 2, 4: None}}
```

sage.graphs.base.boost_graph.**min_cycle_basis**(*g_sage*, *weight_function=None*, *by_weight=False*)

Return a minimum weight cycle basis of the input graph `g_sage`.

A cycle basis is a list of cycles (list of vertices forming a cycle) of `g_sage`. Note that the vertices are not necessarily returned in the order in which they appear in the cycle.

A minimum weight cycle basis is a cycle basis that minimizes the sum of the weights (length for unweighted graphs) of its cycles.

Not implemented for directed graphs and multigraphs.

INPUT:

- `g_sage` – a Sage Graph
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1

EXAMPLES:

```
sage: g = Graph([(1, 2, 3), (2, 3, 5), (3, 4, 8), (4, 1, 13), (1, 3, 250), (5, 6,␣
↪9), (6, 7, 17), (7, 5, 20)])
sage: sorted(g.minimum_cycle_basis(by_weight=True))
[[1, 2, 3], [1, 2, 3, 4], [5, 6, 7]]
sage: sorted(g.minimum_cycle_basis())
[[1, 2, 3], [1, 3, 4], [5, 6, 7]]
```

See also:

- Wikipedia article Cycle_basis

sage.graphs.base.boost_graph.**min_spanning_tree**(*g*, *weight_function=None*, *algorithm='Kruskal'*)

Use Boost to compute the minimum spanning tree of the input graph.

INPUT:

- `g` – the input Sage graph
- `weight_function` – function (default: `None`); a function that inputs an edge `e` and outputs its weight. An edge has the form `(u,v,l)`, where `u` and `v` are vertices, `l` is a label (that can be of any kind). The `weight_function` can be used to transform the label into a weight (see the example below). In particular:
  - if `weight_function` is not `None`, the weight of an edge `e` is `weight_function(e)`;
  - if `weight_function` is `None` (default) and `g` is weighted (that is, `g.weighted()==True`), for each edge `e=(u,v,l)`, we set weight `l`;
  - if `weight_function` is `None` and `g` is not weighted, we set all weights to 1 (hence, the output can be any spanning tree).

  Note that, if the weight is not convertible to a number with function `float()`, an error is raised (see tests below).
- `algorithm` – string (default: `'Kruskal'`); the algorithm to use among `'Kruskal'` and `'Prim'`

OUTPUT:

The edges of a minimum spanning tree of `g`, if one exists, otherwise the empty list.

See also:

- *sage.graphs.generic_graph.GenericGraph.min_spanning_tree()*

EXAMPLES:

```
sage: from sage.graphs.base.boost_graph import min_spanning_tree
sage: min_spanning_tree(graphs.PathGraph(4))
[(0, 1, None), (1, 2, None), (2, 3, None)]

sage: G = Graph([(0,1,{'name':'a','weight':1}), (0,2,{'name':'b','weight':3}), (1,
↪2,{'name':'b','weight':1})])
sage: min_spanning_tree(G, weight_function=lambda e: e[2]['weight'])
[(0, 1, {'name': 'a', 'weight': 1}), (1, 2, {'name': 'b', 'weight': 1})]
```

sage.graphs.base.boost_graph.**shortest_paths**(*g*, *start*, *weight_function=None*, *algorithm=None*)

Compute the shortest paths from `start` to all other vertices.

This routine outputs all shortest paths from node `start` to any other node in the graph. The input graph can be weighted: if the algorithm is Dijkstra, no negative weights are allowed, while if the algorithm is Bellman-Ford, negative weights are allowed, but there must be no negative cycle (otherwise, the shortest paths might not exist).

However, Dijkstra algorithm is more efficient: for this reason, we suggest to use Bellman-Ford only if necessary (which is also the default option). Note that, if the graph is undirected, a negative edge automatically creates a negative cycle: for this reason, in this case, Dijkstra algorithm is always better.

The running-time is $O(n \log n + m)$ for Dijkstra algorithm and $O(mn)$ for Bellman-Ford algorithm, where n is the number of nodes and m is the number of edges.

INPUT:

- `g` – the input Sage graph
- `start` – the starting vertex to compute shortest paths
- `weight_function` – function (default: `None`); a function that associates a weight to each edge. If `None` (default), the weights of `g` are used, if available, otherwise all edges have weight 1.
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'Dijkstra'`, `'Dijkstra_Boost'`: the Dijkstra algorithm implemented in Boost (works only with positive weights)
  - `'Bellman-Ford'`, `'Bellman-Ford_Boost'`: the Bellman-Ford algorithm implemented in Boost (works also with negative weights, if there is no negative cycle)

OUTPUT:

A pair of dictionaries `(distances, predecessors)` such that, for each vertex `v`, `distances[v]` is the distance from `start` to `v`, `predecessors[v]` is the last vertex in a shortest path from `start` to `v`.

EXAMPLES:

Undirected graphs:

```
sage: from sage.graphs.base.boost_graph import shortest_paths
sage: g = Graph([(0,1,1),(1,2,2),(1,3,4),(2,3,1)], weighted=True)
sage: shortest_paths(g, 1)
({0: 1, 1: 0, 2: 2, 3: 3}, {0: 1, 1: None, 2: 1, 3: 2})
sage: g = graphs.GridGraph([2,2])
sage: shortest_paths(g,(0,0),weight_function=lambda e:2)
({(0, 0): 0, (0, 1): 2, (1, 0): 2, (1, 1): 4},
 {(0, 0): None, (0, 1): (0, 0), (1, 0): (0, 0), (1, 1): (0, 1)})
```

Directed graphs:

```
sage: g = DiGraph([(0,1,1),(1,2,2),(1,3,4),(2,3,1)], weighted=True)
sage: shortest_paths(g, 1)
({1: 0, 2: 2, 3: 3}, {1: None, 2: 1, 3: 2})
```

CHAPTER

FOUR

HYPERGRAPHS

4.1 Hypergraph generators

This module implements generators of hypergraphs. All hypergraphs can be built through the `hypergraphs` object. For instance, to build a complete 3-uniform hypergraph on 5 points, one can do:

```
sage: H = hypergraphs.CompleteUniform(5, 3)
```

To enumerate hypergraphs with certain properties up to isomorphism, one can use method `nauty()`, which calls Brendan McKay's Nauty (http://cs.anu.edu.au/~bdm/nauty/):

```
sage: list(hypergraphs.nauty(2, 2, connected=True))
[((0,), (0, 1))]
```

This module contains the following hypergraph generators

nauty()	Enumerate hypergraphs up to isomorphism using Nauty.
CompleteUniform()	Return the complete k-uniform hypergraph on n points.
UniformRandomUniform()	Return a uniformly sampled k-uniform hypergraph on n points with m hyperedges.

4.1.1 Functions and methods

class sage.graphs.hypergraph_generators.**HypergraphGenerators**

Bases: object

A class consisting of constructors for common hypergraphs.

BinomialRandomUniform(n, k, p)

Return a random k-uniform hypergraph on n points, in which each edge is inserted independently with probability p.

- n – number of nodes of the graph
- k – uniformity
- p – probability of an edge

EXAMPLES:

```
sage: hypergraphs.BinomialRandomUniform(50, 3, 1).num_blocks()
19600
```

(continues on next page)

(continued from previous page)

```
sage: hypergraphs.BinomialRandomUniform(50, 3, 0).num_blocks()
0
```

CompleteUniform(n, k)

Return the complete k-uniform hypergraph on n points.

INPUT:

- `k, n` – nonnegative integers with $k \leq n$

EXAMPLES:

```
sage: h = hypergraphs.CompleteUniform(5, 2); h
Incidence structure with 5 points and 10 blocks
sage: len(h.packing())
2
```

UniformRandomUniform(n, k, m)

Return a uniformly sampled k-uniform hypergraph on n points with m hyperedges.

- `n` – number of nodes of the graph
- `k` – uniformity
- `m` – number of edges

EXAMPLES:

```
sage: H = hypergraphs.UniformRandomUniform(52, 3, 17)
sage: H
Incidence structure with 52 points and 17 blocks
sage: H.is_connected()
False
```

nauty(*number_of_sets*, *number_of_vertices*, *multiple_sets=False*, *vertex_min_degree=None*, *vertex_max_degree=None*, *set_max_size=None*, *set_min_size=None*, *regular=False*, *uniform=False*, *max_intersection=None*, *connected=False*, *debug=False*, *options=''*)

Enumerate hypergraphs up to isomorphism using Nauty.

INPUT:

- `number_of_sets`, `number_of_vertices` – integers.
- `multiple_sets` – boolean (default: `False`); whether to allow several sets of the hypergraph to be equal.
- `vertex_min_degree`, `vertex_max_degree` – integers (default: `None`); define the maximum and minimum degree of an element from the ground set (i.e. the number of sets which contain it).
- `set_min_size`, `set_max_size` – integers (default: `None`); define the maximum and minimum size of a set.
- `regular` – integers (default: `False`); if set to an integer value k, requires the hypergraphs to be k-regular. It is actually a shortcut for the corresponding min/max values.
- `uniform` – integers (default: `False`); if set to an integer value k, requires the hypergraphs to be k-uniform. It is actually a shortcut for the corresponding min/max values.
- `max_intersection` – integers (default: `None`); constraints the maximum cardinality of the intersection of two sets fro the hypergraphs.
- `connected` – boolean (default: `False`); whether to require the hypergraphs to be connected.

- `debug` – boolean (default: `False`); if `True` the first line of genbg's output to standard error is captured and the first call to the generator's `next()` function will return this line as a string. A line leading with ">A" indicates a successful initiation of the program with some information on the arguments, while a line beginning with ">E" indicates an error with the input.
- `options` – string (default: `""`) – anything else that should be forwarded as input to Nauty's genbg. See its documentation for more information : http://cs.anu.edu.au/~bdm/nauty/.

Note: For genbg the *first class* elements are vertices, and *second class* elements are the hypergraph's sets.

OUTPUT:

A tuple of tuples.

EXAMPLES:

Small hypergraphs:

```
sage: list(hypergraphs.nauty(4, 2))
[((), (0,), (1,), (0, 1))]
```

Only connected ones:

```
sage: list(hypergraphs.nauty(2, 2, connected=True))
[((0,), (0, 1))]
```

Non-empty sets only:

```
sage: list(hypergraphs.nauty(3, 2, set_min_size=1))
[((0,), (1,), (0, 1))]
```

The Fano Plane, as the only 3-uniform hypergraph with 7 sets and 7 vertices:

```
sage: fano = next(hypergraphs.nauty(7, 7, uniform=3, max_intersection=1))
sage: print(fano)
((0, 1, 2), (0, 3, 4), (0, 5, 6), (1, 3, 5), (2, 4, 5), (2, 3, 6), (1, 4, 6))
```

The Fano Plane, as the only 3-regular hypergraph with 7 sets and 7 vertices:

```
sage: fano = next(hypergraphs.nauty(7, 7, regular=3, max_intersection=1))
sage: print(fano)
((0, 1, 2), (0, 3, 4), (0, 5, 6), (1, 3, 5), (2, 4, 5), (2, 3, 6), (1, 4, 6))
```

4.2 Incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure is specified by a list of points, blocks, or an incidence matrix ([1],[2]). *IncidenceStructure* instances have the following methods:

[1] Block designs and incidence structures from wikipedia, Wikipedia article Block_design Wikipedia article Incidence_structure
[2] E. Assmus, J. Key, Designs and their codes, CUP, 1992.

automorphism_group()	Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.
block_sizes()	Return the set of block sizes.
blocks()	Return the list of blocks.
canonical_label()	Return a canonical label for the incidence structure.
coloring()	Compute a (weak) k-coloring of the hypergraph
complement()	Return the complement of the incidence structure.
copy()	Return a copy of the incidence structure.
degree()	Return the degree of a point `p` (or a set of points).
degrees()	Return the degree of all sets of given size, or the degree of all points.
dual()	Return the dual of the incidence structure.
edge_coloring()	Compute a proper edge-coloring.
ground_set()	Return the ground set (i.e the list of points).
incidence_graph()	Return the incidence graph of the incidence structure
incidence_matrix()	Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.
induced_substructure()	Return the substructure induced by a set of points.
intersection_graph()	Return the intersection graph of the incidence structure.
is_berge_cyclic()	Check whether `self` is a Berge-Cyclic uniform hypergraph.
is_connected()	Test whether the design is connected.
is_generalized_quadrangle()	Test if the incidence structure is a generalized quadrangle.
is_isomorphic()	Return whether the two incidence structures are isomorphic.
is_regular()	Test whether the incidence structure is r-regular.
is_resolvable()	Test whether the hypergraph is resolvable
is_simple()	Test whether this design is simple (i.e. no repeated block).
is_t_design()	Test whether `self` is a $t-(v,k,l)$ design.
is_uniform()	Test whether the incidence structure is k-uniform
isomorphic_substructures_iterator()	Iterates over all copies of `H2` contained in `self`.
num_blocks()	Return the number of blocks.
num_points()	Return the size of the ground set.
packing()	Return a maximum packing
rank()	Return the rank of the hypergraph (the maximum size of a block).
relabel()	Relabel the ground set
trace()	Return the trace of a set of points.

REFERENCES:

AUTHORS:

- Peter Dobcsanyi and David Joyner (2007-2008)

  This is a significantly modified form of part of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org.

- Vincent Delecroix (2014): major rewrite

4.2.1 Methods

class sage.combinat.designs.incidence_structures.**IncidenceStructure**(*points=None*, *blocks=None*, *incidence_matrix=None*, *name=None*, *check=True*, *copy=True*)

Bases: `object`

A base class for incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure (i.e. hypergraph, i.e. set system) can be defined from a collection of blocks (i.e. sets, i.e. edges), optionally with an explicit ground set (i.e. point set, i.e. vertex set). Alternatively they can be defined from a binary incidence matrix.

INPUT:

- `points` – (i.e. ground set, i.e. vertex set) the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$.

  Note: The following syntax, where `points` is ommitted, automatically defines the ground set as the union of the blocks:

  ```
  sage: H = IncidenceStructure([['a','b','c'],['c','d','e']])
  sage: sorted(H.ground_set())
  ['a', 'b', 'c', 'd', 'e']
  ```

- `blocks` – (i.e. edges, i.e. sets) the blocks defining the incidence structure. Can be any iterable.
- `incidence_matrix` – a binary incidence matrix. Each column represents a set.
- `name` (a string, such as "Fano plane").
- `check` – whether to check the input
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the *`IncidenceStructure`* instance's internal data.

EXAMPLES:

An incidence structure can be constructed by giving the number of points and the list of blocks:

```
sage: IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,
↪5]])
Incidence structure with 7 points and 7 blocks
```

Only providing the set of blocks is sufficient. In this case, the ground set is defined as the union of the blocks:

```
sage: IncidenceStructure([[1,2,3],[2,3,4]])
Incidence structure with 4 points and 2 blocks
```

Or by its adjacency matrix (a $\{0,1\}$-matrix in which rows are indexed by points and columns by blocks):

```
sage: m = matrix([[0,1,0],[0,0,1],[1,0,1],[1,1,1]])
sage: IncidenceStructure(m)
Incidence structure with 4 points and 3 blocks
```

The points can be any (hashable) object:

```
sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [(V[0],V[1],V[2]), (V[1],V[2]), (V[0],V[2])]
sage: I = IncidenceStructure(V, B)
sage: I.ground_set()
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b')]
sage: I.blocks()
[[(0, 'a'), (0, 'b'), (1, 'a')], [(0, 'a'), (1, 'a')], [(0, 'b'), (1, 'a')]]
```

The order of the points and blocks does not matter as they are sorted on input (see trac ticket #11333):

```
sage: A = IncidenceStructure([0,1,2], [[0],[0,2]])
sage: B = IncidenceStructure([1,0,2], [[0],[2,0]])
sage: B == A
True

sage: C = BlockDesign(2, [[0], [1,0]])
sage: D = BlockDesign(2, [[0,1], [0]])
sage: C == D
True
```

If you care for speed, you can set `copy` to `False`, but in that case, your input must be a list of lists and the ground set must be $0, ..., v-1$:

```
sage: blocks = [[0,1],[2,0],[1,2]]  # a list of lists of integers
sage: I = IncidenceStructure(3, blocks, copy=False)
sage: I._blocks is blocks
True
```

automorphism_group()

Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.

EXAMPLES:

```
sage: P = designs.DesarguesianProjectivePlaneDesign(2); P
(7,3,1)-Balanced Incomplete Block Design
sage: G = P.automorphism_group()
sage: G.is_isomorphic(PGL(3,2))
True
sage: G
Permutation Group with generators [...]
sage: G.cardinality()
168
```

A non self-dual example:

```
sage: IS = IncidenceStructure(list(range(4)), [[0,1,2,3],[1,2,3]])
sage: IS.automorphism_group().cardinality()
6
sage: IS.dual().automorphism_group().cardinality()
1
```

Examples with non-integer points:

```
sage: I = IncidenceStructure('abc', ('ab','ac','bc'))
sage: I.automorphism_group()
```

(continues on next page)

(continued from previous page)

```
Permutation Group with generators [('b','c'), ('a','b')]
sage: IncidenceStructure([[(1,2),(3,4)]]).automorphism_group()
Permutation Group with generators [((1,2),(3,4))]
```

block_sizes()

Return the set of block sizes.

EXAMPLES:

```
sage: BD = IncidenceStructure(8, [[0,1,3],[1,4,5,6],[1,2],[5,6,7]])
sage: BD.block_sizes()
[3, 2, 4, 3]
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,
↪6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

blocks()

Return the list of blocks.

EXAMPLES:

```
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,
↪6],[2,4,5]])
sage: BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

canonical_label()

Return a canonical label for the incidence structure.

A canonical label is relabeling of the points into integers $\{0, ..., n-1\}$ such that isomorphic incidence structures are relabelled to equal objects.

EXAMPLES:

```
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1 == fano2
False
sage: fano1.relabel(fano1.canonical_label())
sage: fano2.relabel(fano2.canonical_label())
sage: fano1 == fano2
True
```

coloring(*k=None*, *solver=None*, *verbose=0*)

Compute a (weak) k-coloring of the hypergraph

A weak coloring of a hypergraph $\mathcal{H}$ is an assignment of colors to its vertices such that no set is monochromatic.

INPUT:

- `k` (integer) – compute a coloring with k colors if an integer is provided, otherwise returns an optimal coloring (i.e. with the minimum possible number of colors).
- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.

- verbose – non-negative integer (default: 0). Set the level of verbosity you want from the linear program solver. Since the problem is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver.

EXAMPLES:

The Fano plane has chromatic number 3:

```
sage: len(designs.steiner_triple_system(7).coloring())
3
```

One admissible 3-coloring:

```
sage: designs.steiner_triple_system(7).coloring() # not tested - architecture-
↪dependent
[[0, 2, 5, 1], [4, 3], [6]]
```

The chromatic number of a graph is equal to the chromatic number of its 2-uniform corresponding hypergraph:

```
sage: g = graphs.PetersenGraph()
sage: H = IncidenceStructure(g.edges(labels=False))
sage: len(g.coloring())
3
sage: len(H.coloring())
3
```

complement (*uniform=False*)

Return the complement of the incidence structure.

Two different definitions of "complement" are made available, according to the value of uniform.

INPUT:

- uniform (boolean) –
  - if set to False (default), returns the incidence structure whose blocks are the complements of all blocks of the incidence structure.
  - If set to True and the incidence structure is k-uniform, returns the incidence structure whose blocks are all k-sets of the ground set that do not appear in self.

EXAMPLES:

The complement of a BalancedIncompleteBlockDesign is also a 2-design:

```
sage: bibd = designs.balanced_incomplete_block_design(13,4)
sage: bibd.is_t_design(return_parameters=True)
(True, (2, 13, 4, 1))
sage: bibd.complement().is_t_design(return_parameters=True)
(True, (2, 13, 9, 6))
```

The "uniform" complement of a graph is a graph:

```
sage: g = graphs.PetersenGraph()
sage: G = IncidenceStructure(g.edges(labels=False))
sage: H = G.complement(uniform=True)
sage: h = Graph(H.blocks())
sage: g == h
False
```

(continues on next page)

(continued from previous page)

```
sage: g == h.complement()
True
```

copy()

Return a copy of the incidence structure.

EXAMPLES:

```
sage: IS = IncidenceStructure([[1,2,3,"e"]],name="Test")
sage: IS
Incidence structure with 4 points and 1 blocks
sage: copy(IS)
Incidence structure with 4 points and 1 blocks
sage: [1, 2, 3, 'e'] in copy(IS)
True
sage: copy(IS)._name
'Test'
```

degree(*p=None*, *subset=False*)

Return the degree of a point `p` (or a set of points).

The degree of a point (or set of points) is the number of blocks that contain it.

INPUT:

- `p` – a point (or a set of points) of the incidence structure.
- `subset` (boolean) – whether to interpret the argument as a set of point (`subset=True`) or as a point (`subset=False`, default).

EXAMPLES:

```
sage: designs.steiner_triple_system(9).degree(3)
4
sage: designs.steiner_triple_system(9).degree({1,2},subset=True)
1
```

degrees(*size=None*)

Return the degree of all sets of given size, or the degree of all points.

The degree of a point (or set of point) is the number of blocks that contain it.

INPUT:

- `size` (integer) – return the degree of all subsets of points of cardinality `size`. When `size=None`, the function outputs the degree of all points.

  Note: When `size=None` the output is indexed by the points. When `size=1` it is indexed by tuples of size 1. This is the same information, stored slightly differently.

OUTPUT:

A dictionary whose values are degrees and keys are either:

- the points of the incidence structure if `size=None` (default)
- the subsets of size `size` of the points stored as tuples

EXAMPLES:

```
sage: IncidenceStructure([[1,2,3],[1,4]]).degrees(2)
{(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 3): 1, (2, 4): 0, (3, 4): 0}
```

In a Steiner triple system, all pairs have degree 1:

```
sage: S13 = designs.steiner_triple_system(13)
sage: all(v == 1 for v in S13.degrees(2).values())
True
```

dual(*algorithm=None*)

Return the dual of the incidence structure.

INPUT:

- `algorithm` – whether to use Sage's implementation (`algorithm=None`, default) or use GAP's (`algorithm="gap"`).

Note: The `algorithm="gap"` option requires GAP's Design package (included in the gap_packages Sage spkg).

EXAMPLES:

The dual of a projective plane is a projective plane:

```
sage: PP = designs.DesarguesianProjectivePlaneDesign(4)
sage: PP.dual().is_t_design(return_parameters=True)
(True, (2, 21, 5, 1))
```

REFERENCE:

- Soicher, Leonard, Design package manual, available at http://www.gap-system.org/Manuals/pkg/design/htm/CHAP003.htm

edge_coloring()

Compute a proper edge-coloring.

A proper edge-coloring is an assignment of colors to the sets of the incidence structure such that two sets with non-empty intersection receive different colors. The coloring returned minimizes the number of colors.

OUTPUT:

A partition of the sets into color classes.

EXAMPLES:

```
sage: H = Hypergraph([{1,2,3},{2,3,4},{3,4,5},{4,5,6}]); H
Incidence structure with 6 points and 4 blocks
sage: C = H.edge_coloring()
sage: C # random
[[[3, 4, 5]], [[2, 3, 4]], [[4, 5, 6], [1, 2, 3]]]
sage: Set(map(Set,sum(C,[]))) == Set(map(Set,H.blocks()))
True
```

ground_set()

Return the ground set (i.e the list of points).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).ground_set()
[0, 1, 2]
```

incidence_graph(*labels=False*)

Return the incidence graph of the incidence structure

A point and a block are adjacent in this graph whenever they are incident.

INPUT:

- `labels` (boolean) – whether to return a graph whose vertices are integers, or labelled elements.
  - `labels is False` (default) – in this case the first vertices of the graphs are the elements of *`ground_set()`*, and appear in the same order. Similarly, the following vertices represent the elements of *`blocks()`*, and appear in the same order.
  - `labels is True`, the points keep their original labels, and the blocks are `Set` objects.

    Note that the labelled incidence graph can be incorrect when blocks are repeated, and on some (rare) occasions when the elements of *`ground_set()`* mix `Set()` and non-`Set` objects.

EXAMPLES:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,
↪3,6],[2,4,5]])
sage: BD.incidence_graph()
Bipartite graph on 14 vertices
sage: A = BD.incidence_matrix()
sage: Graph(block_matrix([[A*0,A],[A.transpose(),A*0]])) == BD.incidence_
↪graph()
True
```

incidence_matrix()

Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.

EXAMPLES:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,
↪3,6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
sage: BD.incidence_matrix()
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]

sage: I = IncidenceStructure('abc', ('ab','abc','ac','c'))
sage: I.incidence_matrix()
[1 1 1 0]
[1 1 0 0]
[0 1 1 1]
```

induced_substructure(*points*)

Return the substructure induced by a set of points.

The substructure induced in $\mathcal{H}$ by a set $X \subseteq V(\mathcal{H})$ of points is the incidence structure $\mathcal{H}_X$ defined on X whose sets are all $S \in \mathcal{H}$ such that $S \subseteq X$.

INPUT:

- `points` – a set of points.

Note: This method goes over all sets of `self` before building a new *`IncidenceStructure`* (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLES:

A Fano plane with one point removed:

```
sage: F = designs.steiner_triple_system(7)
sage: F.induced_substructure([0..5])
Incidence structure with 6 points and 4 blocks
```

intersection_graph(*sizes=None*)

Return the intersection graph of the incidence structure.

The vertices of this graph are the *`blocks()`* of the incidence structure. Two of them are adjacent if the size of their intersection belongs to the set `sizes`.

INPUT:

- `sizes` – a list/set of integers. For convenience, setting `sizes` to `5` has the same effect as `sizes=[5]`. When set to `None` (default), behaves as `sizes=PositiveIntegers()`.

EXAMPLES:

The intersection graph of a `balanced_incomplete_block_design()` is a *`strongly regular graph`* (when it is not trivial):

```
sage: BIBD =  designs.balanced_incomplete_block_design(19,3)
sage: G = BIBD.intersection_graph(1)
sage: G.is_strongly_regular(parameters=True)
(57, 24, 11, 9)
```

is_berge_cyclic()

Check whether `self` is a Berge-Cyclic uniform hypergraph.

A k-uniform Berge cycle (named after Claude Berge) of length ℓ is a cyclic list of distinct k-sets $F_1, \ldots, F_\ell$, $\ell > 1$, and distinct vertices $C = \{v_1, \ldots, v_\ell\}$ such that for each $1 \leq i \leq \ell$, F_i contains v_i and v_{i+1} (where $v_{l+1} = v_1$).

A uniform hypergraph is Berge-cyclic if its incidence graph is cyclic. It is called "Berge-acyclic" otherwise.

For more information, see [Fag1983] and Wikipedia article Hypergraph.

EXAMPLES:

```
sage: Hypergraph(5, [[1, 2, 3], [2, 3 ,4]]).is_berge_cyclic()
True
sage: Hypergraph(6, [[1, 2, 3], [3 ,4, 5]]).is_berge_cyclic()
False
```

is_connected()

Test whether the design is connected.

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).is_connected()
True
sage: IncidenceStructure(4, [[0,1],[2,3]]).is_connected()
False
```

is_generalized_quadrangle(*verbose=False*, *parameters=False*)

Test if the incidence structure is a generalized quadrangle.

An incidence structure is a generalized quadrangle iff (see [**?**], section 9.6):

- two blocks intersect on at most one point.
- For every point p not in a block B, there is a unique block B' intersecting both $\{p\}$ and B

It is a *regular* generalized quadrangle if furthermore:

- it is $s+1$-*uniform* for some positive integer s.
- it is $t+1$-*regular* for some positive integer t.

For more information, see the Wikipedia article Generalized_quadrangle.

Note: Some references (e.g. [PT2009] or Wikipedia article Generalized_quadrangle) only allow *regular* generalized quadrangles. To use such a definition, see the `parameters` optional argument described below, or the methods *is_regular()* and *is_uniform()*.

INPUT:

- `verbose` (boolean) – whether to print an explanation when the instance is not a generalized quadrangle.
- `parameters` (boolean; `False`) – if set to `True`, the function returns a pair `(s,t)` instead of `True` answers. In this case, s and t are the integers defined above if they exist (each can be set to `False` otherwise).

EXAMPLES:

```
sage: h = designs.CremonaRichmondConfiguration()
sage: h.is_generalized_quadrangle()
True
```

This is actually a *regular* generalized quadrangle:

```
sage: h.is_generalized_quadrangle(parameters=True)
(2, 2)
```

is_isomorphic(*other*, *certificate=False*)

Return whether the two incidence structures are isomorphic.

INPUT:

- `other` – an incidence structure.
- `certificate` (boolean) – whether to return an isomorphism from `self` to `other` instead of a boolean answer.

EXAMPLES:

```
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1.is_isomorphic(fano2)
True
sage: fano1.is_isomorphic(fano2,certificate=True)
{0: 0, 1: 1, 2: 2, 3: 6, 4: 4, 5: 3, 6: 5}
```

is_regular(*r=None*)

Test whether the incidence structure is r-regular.

An incidence structure is said to be r-regular if all its points are incident with exactly r blocks.

INPUT:

- `r` (integer)

OUTPUT:

If `r` is defined, a boolean is returned. If `r` is set to `None` (default), the method returns either `False` or the integer `r` such that the incidence structure is r-regular.

Warning: In case of 0-regular incidence structure, beware that `if not H.is_regular()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_regular()
3
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=3)
True
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=4)
False
```

is_resolvable(*certificate=False*, *solver=None*, *verbose=0*, *check=True*)

Test whether the hypergraph is resolvable

A hypergraph is said to be resolvable if its sets can be partitionned into classes, each of which is a partition of the ground set.

Note: This problem is solved using an Integer Linear Program, and GLPK (the default LP solver) has been reported to be very slow on some instances. If you hit this wall, consider installing a more powerful LP solver (CPLEX, Gurobi, ...).

INPUT:

- `certificate` (boolean) – whether to return the classes along with the binary answer (see examples below).
- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

Some resolvable designs:

```
sage: TD = designs.transversal_design(2,2,resolvable=True)
sage: TD.is_resolvable()
True

sage: AG = designs.AffineGeometryDesign(3,1,GF(2))
sage: AG.is_resolvable()
True
```

Their classes:

```
sage: b,cls = TD.is_resolvable(True)
sage: b
True
sage: cls # random
[[[0, 3], [1, 2]], [[1, 3], [0, 2]]]

sage: b,cls = AG.is_resolvable(True)
sage: b
True
sage: cls # random
[[[6, 7], [4, 5], [0, 1], [2, 3]],
 [[5, 7], [0, 4], [3, 6], [1, 2]],
 [[0, 2], [4, 7], [1, 3], [5, 6]],
 [[3, 4], [0, 7], [1, 5], [2, 6]],
 [[3, 7], [1, 6], [0, 5], [2, 4]],
 [[0, 6], [2, 7], [1, 4], [3, 5]],
 [[4, 6], [0, 3], [2, 5], [1, 7]]]
```

A non-resolvable design:

```
sage: Fano = designs.balanced_incomplete_block_design(7,3)
sage: Fano.is_resolvable()
False
sage: Fano.is_resolvable(True)
(False, [])
```

is_simple()

Test whether this design is simple (i.e. no repeated block).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[1,2],[0,2]]).is_simple()
True
sage: IncidenceStructure(3, [[0],[0]]).is_simple()
False

sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [[V[0],V[1]], [V[1],V[2]]]
sage: I = IncidenceStructure(V, B)
sage: I.is_simple()
True
sage: I2 = IncidenceStructure(V, B*2)
sage: I2.is_simple()
False
```

is_t_design(*t=None*, *v=None*, *k=None*, *l=None*, *return_parameters=False*)

Test whether `self` is a $t-(v,k,l)$ design.

A $t-(v,k,\lambda)$ (sometimes called t-design for short) is a block design in which:

- the underlying set has cardinality v
- the blocks have size k
- each t-subset of points is covered by λ blocks

INPUT:

- `t,v,k,l` (integers) – their value is set to `None` by default. The function tests whether the design is a `t-(v,k,l)` design using the provided values and guesses the others. Note that l cannot be specified if `t` is not.
- `return_parameters` (boolean)– whether to return the parameters of the t-design. If set to `True`, the function returns a pair `(boolean_answer,(t,v,k,l))`.

EXAMPLES:

```
sage: fano_blocks = [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
sage: BD = IncidenceStructure(7, fano_blocks)
sage: BD.is_t_design()
True
sage: BD.is_t_design(return_parameters=True)
(True, (2, 7, 3, 1))
sage: BD.is_t_design(2, 7, 3, 1)
True
sage: BD.is_t_design(1, 7, 3, 3)
True
sage: BD.is_t_design(0, 7, 3, 7)
True

sage: BD.is_t_design(0,6,3,7) or BD.is_t_design(0,7,4,7) or BD.is_t_design(0,
↪7,3,8)
False

sage: BD = designs.AffineGeometryDesign(3, 1, GF(2))
sage: BD.is_t_design(1)
True
sage: BD.is_t_design(2)
True
```

Steiner triple and quadruple systems are other names for $2-(v,3,1)$ and $3-(v,4,1)$ designs:

```
sage: S3_9 = designs.steiner_triple_system(9)
sage: S3_9.is_t_design(2,9,3,1)
True

sage: blocks = designs.steiner_quadruple_system(8)
sage: S4_8 = IncidenceStructure(8, blocks)
sage: S4_8.is_t_design(3,8,4,1)
True

sage: blocks = designs.steiner_quadruple_system(14)
sage: S4_14 = IncidenceStructure(14, blocks)
sage: S4_14.is_t_design(3,14,4,1)
True
```

Some examples of Witt designs that need the gap database:

```
sage: BD = designs.WittDesign(9)          # optional - gap_packages
sage: BD.is_t_design(2,9,3,1)             # optional - gap_packages
True
sage: W12 = designs.WittDesign(12)        # optional - gap_packages
sage: W12.is_t_design(5,12,6,1)           # optional - gap_packages
True
sage: W12.is_t_design(4)                  # optional - gap_packages
True
```

Further examples:

```
sage: D = IncidenceStructure(4,[[],[]])
sage: D.is_t_design(return_parameters=True)
(True,  (0, 4, 0, 2))

sage: D = IncidenceStructure(4, [[0,1],[0,2],[0,3]])
sage: D.is_t_design(return_parameters=True)
(True, (0, 4, 2, 3))

sage: D = IncidenceStructure(4, [[0],[1],[2],[3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 1, 1))

sage: D = IncidenceStructure(4,[[0,1],[2,3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 2, 1))

sage: D = IncidenceStructure(4, [list(range(4))])
sage: D.is_t_design(return_parameters=True)
(True, (4, 4, 4, 1))
```

is_uniform(*k=None*)

Test whether the incidence structure is k-uniform

An incidence structure is said to be k-uniform if all its blocks have size k.

INPUT:

- `k` (integer)

OUTPUT:

If `k` is defined, a boolean is returned. If `k` is set to `None` (default), the method returns either `False` or the integer `k` such that the incidence structure is k-uniform.

Warning: In case of 0-uniform incidence structure, beware that `if not H.is_uniform()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_uniform()
3
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=3)
True
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=4)
False
```

isomorphic_substructures_iterator(*H2*, *induced=False*)

Iterates over all copies of `H2` contained in `self`.

A hypergraph H_1 contains an isomorphic copy of a hypergraph H_2 if there exists an injection $f : V(H_2) \mapsto V(H_1)$ such that for any set $S_2 \in E(H_2)$ the set $S_1 = f(S2)$ belongs to $E(H_1)$.

It is an *induced* copy if no other set of $E(H_1)$ is contained in $f(V(H_2))$, i.e. $|E(H_2)| = \{S : S \in E(H_1)$ and $f(V(H_2))\}$.

This function lists all such injections. In particular, the number of copies of H in itself is equal to *the size of its automorphism group*.

See `subhypergraph_search` for more information.

INPUT:

- `H2` an *IncidenceStructure* object.
- `induced` (boolean) – whether to require the copies to be induced. Set to `False` by default.

EXAMPLES:

How many distinct C_5 in Petersen's graph ?

```
sage: P = graphs.PetersenGraph()
sage: C = graphs.CycleGraph(5)
sage: IP = IncidenceStructure(P.edges(labels=False))
sage: IC = IncidenceStructure(C.edges(labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
120
```

As the automorphism group of C_5 has size 10, the number of distinct unlabelled copies is 12. Let us check that all functions returned correspond to an actual C_5 subgraph:

```
sage: for f in IP.isomorphic_substructures_iterator(IC):
....:     assert all(P.has_edge(f[x],f[y]) for x,y in C.edges(labels=False))
```

The number of induced copies, in this case, is the same:

```
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
120
```

They begin to differ if we make one vertex universal:

```
sage: P.add_edges([(0,x) for x in P], loops=False)
sage: IP = IncidenceStructure(P.edges(labels=False))
sage: IC = IncidenceStructure(C.edges(labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
420
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
60
```

The number of copies of H in itself is the size of its automorphism group:

```
sage: H = designs.projective_plane(3)
sage: sum(1 for _ in H.isomorphic_substructures_iterator(H))
5616
sage: H.automorphism_group().cardinality()
5616
```

num_blocks()
Return the number of blocks.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_blocks()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_blocks()
5
```

num_points()
Return the size of the ground set.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_points()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_points()
4
```

packing(*solver=None*, *verbose=0*)
Return a maximum packing

A maximum packing in a hypergraph is collection of disjoint sets/blocks of maximal cardinality. This problem is NP-complete in general, and in particular on 3-uniform hypergraphs. It is solved here with an Integer Linear Program.

For more information, see the Wikipedia article Packing_in_a_hypergraph.

INPUT:

- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: P = IncidenceStructure([[1,2],[3,4],[2,3]]).packing()
sage: sorted(sorted(b) for b in P)
[[1, 2], [3, 4]]
sage: len(designs.steiner_triple_system(9).packing())
3
```

rank()
Return the rank of the hypergraph (the maximum size of a block).

EXAMPLES:

```
sage: h = Hypergraph(8, [[0,1,3],[1,4,5,6],[1,2]])
sage: h.rank()
4
```

relabel(*perm=None*, *inplace=True*)
Relabel the ground set

INPUT:

- `perm` – can be one of

- a dictionary – then each point `p` (which should be a key of `d`) is relabeled to `d[p]`
- a list or a tuple of length `n` – the first point returned by *ground_set()* is relabeled to `l[0]`, the second to `l[1]`, ...
- `None` – the incidence structure is relabeled to be on $\{0, 1, ..., n-1\}$ in the ordering given by *ground_set()*.

- `inplace` – If `True` then return a relabeled graph and does not touch `self` (default is `False`).

EXAMPLES:

```
sage: TD=designs.transversal_design(5,5)
sage: TD.relabel({i:chr(97+i) for i in range(25)})
sage: TD.ground_set()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p
↪', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
sage: TD.blocks()[:3]
[['a', 'f', 'k', 'p', 'u'], ['a', 'g', 'm', 's', 'y'], ['a', 'h', 'o', 'q', 'x
↪']]
```

Relabel to integer points:

```
sage: TD.relabel()
sage: TD.blocks()[:3]
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23]]
```

trace (*points*, *min_size=1*, *multiset=True*)

Return the trace of a set of points.

Given an hypergraph $\mathcal{H}$, the *trace* of a set X of points in $\mathcal{H}$ is the hypergraph whose blocks are all non-empty $S \cap X$ where $S \in \mathcal{H}$.

INPUT:

- `points` – a set of points.
- `min_size` (integer; default 1) – minimum size of the sets to keep. By default all empty sets are discarded, i.e. `min_size=1`.
- `multiset` (boolean; default `True`) – whether to keep multiple copies of the same set.

Note: This method goes over all sets of `self` before building a new *IncidenceStructure* (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLES:

A Baer subplane of order 2 (i.e. a Fano plane) in a projective plane of order 4:

```
sage: P4 = designs.projective_plane(4)
sage: F = designs.projective_plane(2)
sage: for x in Subsets(P4.ground_set(),7):
....:     if P4.trace(x,min_size=2).is_isomorphic(F):
....:         break
sage: subplane = P4.trace(x,min_size=2); subplane
Incidence structure with 7 points and 7 blocks
sage: subplane.is_isomorphic(F)
True
```

CHAPTER

FIVE

LIBRARIES OF ALGORITHMS

5.1 Graph coloring

This module gathers all methods related to graph coloring. Here is what it can do :

Proper vertex coloring

all_graph_colorings()	Compute all n-colorings a graph
first_coloring()	Return the first vertex coloring found
number_of_n_colorings()	Compute the number of n-colorings of a graph
numbers_of_colorings()	Compute the number of colorings of a graph
chromatic_number()	Return the chromatic number of the graph
vertex_coloring()	Compute vertex colorings and chromatic numbers

Other colorings

grundy_coloring()	Compute Grundy numbers and Grundy colorings
b_coloring()	Compute b-chromatic numbers and b-colorings
edge_coloring()	Compute chromatic index and edge colorings
round_robin()	Compute a round-robin coloring of the complete graph on n vertices
linear_arboricity()	Compute the linear arboricity of the given graph
acyclic_edge_coloring()	Compute an acyclic edge coloring of the current graph

AUTHORS:

- Tom Boothby (2008-02-21): Initial version
- Carlo Hamalainen (2009-03-28): minor change: switch to C++ DLX solver
- Nathann Cohen (2009-10-24): Coloring methods using linear programming

5.1.1 Methods

class sage.graphs.graph_coloring.**Test**

Bases: object

This class performs randomized testing for all_graph_colorings.

Since everything else in this file is derived from all_graph_colorings, this is a pretty good randomized tester for the entire file. Note that for a graph G, `G.chromatic_polynomial()` uses an entirely different algorithm, so we provide a good, independent test.

random(*tests=1000*)
Call `self.random_all_graph_colorings()`.

In the future, if other methods are added, it should call them, too.

random_all_graph_colorings(*tests=2*)
Verify the results of `all_graph_colorings()` in three ways:

1. all colorings are unique
2. number of m-colorings is $P(m)$ (where P is the chromatic polynomial of the graph being tested)
3. colorings are valid – that is, that no two vertices of the same color share an edge.

sage.graphs.graph_coloring.**acyclic_edge_coloring**(*g*, *hex_colors=False*, *value_only=False*, *k=0*, *solver=None*, *verbose=0*)
Compute an acyclic edge coloring of the current graph.

An edge coloring of a graph is a assignment of colors to the edges of a graph such that :

- the coloring is proper (no adjacent edges share a color)
- For any two colors i, j, the union of the edges colored with i or j is a forest.

The least number of colors such that such a coloring exists for a graph G is written $\chi'_a(G)$, also called the acyclic chromatic index of G.

It is conjectured that this parameter can not be too different from the obvious lower bound $\Delta(G) \leq \chi'_a(G)$, $\Delta(G)$ being the maximum degree of G, which is given by the first of the two constraints. Indeed, it is conjectured that $\Delta(G) \leq \chi'_a(G) \leq \Delta(G) + 2$.

INPUT:

- `hex_colors` – boolean (default: `False`):
  - If `hex_colors = True`, the function returns a dictionary associating to each color a list of edges (meant as an argument to the `edge_colors` keyword of the `plot` method).
  - If `hex_colors = False` (default value), returns a list of graphs corresponding to each color class.
- `value_only` – boolean (default: `False`):
  - If `value_only = True`, only returns the acyclic chromatic index as an integer value
  - If `value_only = False`, returns the color classes according to the value of `hex_colors`
- `k` – integer; the number of colors to use.
  - If `k > 0`, computes an acyclic edge coloring using k colors.
  - If `k = 0` (default), computes a coloring of G into $\Delta(G) + 2$ colors, which is the conjectured general bound.
  - If `k = None`, computes a decomposition using the least possible number of colors.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.

ALGORITHM:

Linear Programming

EXAMPLES:

The complete graph on 8 vertices can not be acyclically edge-colored with less $\Delta + 1$ colors, but it can be colored with $\Delta + 2 = 9$:

```
sage: from sage.graphs.graph_coloring import acyclic_edge_coloring
sage: g = graphs.CompleteGraph(8)
sage: colors = acyclic_edge_coloring(g)
```

Each color class is of course a matching

```
sage: all(max(gg.degree()) <= 1 for gg in colors)
True
```

These matchings being a partition of the edge set:

```
sage: all(any(gg.has_edge(e) for gg in colors) for e in g.edge_
↪iterator(labels=False))
True
```

Besides, the union of any two of them is a forest

```
sage: all(g1.union(g2).is_forest() for g1 in colors for g2 in colors)
True
```

If one wants to acyclically color a cycle on 4 vertices, at least 3 colors will be necessary. The function raises an exception when asked to color it with only 2:

```
sage: g = graphs.CycleGraph(4)
sage: acyclic_edge_coloring(g, k=2)
Traceback (most recent call last):
...
ValueError: this graph can not be colored with the given number of colors
```

The optimal coloring give us 3 classes:

```
sage: colors = acyclic_edge_coloring(g, k=None)
sage: len(colors)
3
```

sage.graphs.graph_coloring.**all_graph_colorings**(*G*, *n*, *count_only=False*, *hex_colors=False*, *vertex_color_dict=False*)

Compute all n-colorings of a graph.

This method casts the graph coloring problem into an exact cover problem, and passes this into an implementation of the Dancing Links algorithm described by Knuth (who attributes the idea to Hitotumatu and Noshita).

INPUT:

- `G` – a graph
- `n` – a positive integer; the number of colors
- `count_only` – boolean (default: `False`); when set to `True`, it returns 1 for each coloring
- `hex_colors` – boolean (default: `False`); when set to `False`, colors are labeled [0, 1, ..., $n - 1$], otherwise the RGB Hex labeling is used
- `vertex_color_dict` – boolean (default: `False`); when set to `True`, it returns a dictionary `{vertex: color}`, otherwise it returns a dictionary `{color: [list of vertices]}`

The construction works as follows. Columns:

- The first $|V|$ columns correspond to a vertex – a 1 in this column indicates that that vertex has a color.
- After those $|V|$ columns, we add $n * |E|$ columns – a 1 in these columns indicate that a particular edge is incident to a vertex with a certain color.

Rows:

- For each vertex, add n rows; one for each color c. Place a 1 in the column corresponding to the vertex, and a 1 in the appropriate column for each edge incident to the vertex, indicating that that edge is incident to the color c.
- If $n > 2$, the above construction cannot be exactly covered since each edge will be incident to only two vertices (and hence two colors) - so we add $n * |E|$ rows, each one containing a 1 for each of the $n * |E|$ columns. These get added to the cover solutions "for free" during the backtracking.

Note that this construction results in $n * |V| + 2 * n * |E| + n * |E|$ entries in the matrix. The Dancing Links algorithm uses a sparse representation, so if the graph is simple, $|E| \leq |V|^2$ and $n <= |V|$, this construction runs in $O(|V|^3)$ time. Back-conversion to a coloring solution is a simple scan of the solutions, which will contain $|V| + (n-2) * |E|$ entries, so runs in $O(|V|^3)$ time also. For most graphs, the conversion will be much faster – for example, a planar graph will be transformed for 4-coloring in linear time since $|E| = O(|V|)$.

REFERENCES:

http://www-cs-staff.stanford.edu/~uno/papers/dancing-color.ps.gz

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import all_graph_colorings
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: n = 0
sage: for C in all_graph_colorings(G, 3, hex_colors=True):
....:     parts = [C[k] for k in C]
....:     for P in parts:
....:         l = len(P)
....:         for i in range(l):
....:             for j in range(i + 1, l):
....:                 if G.has_edge(P[i], P[j]):
....:                     raise RuntimeError("Coloring Failed.")
....:     n+=1
sage: print("G has %s 3-colorings." % n)
G has 12 3-colorings.
```

sage.graphs.graph_coloring.**b_coloring**(*g*, *k*, *value_only=True*, *solver=None*, *verbose=0*)

Compute b-chromatic numbers and b-colorings.

This function computes a b-coloring with at most k colors that maximizes the number of colors, if such a coloring exists.

Definition :

Given a proper coloring of a graph G and a color class C such that none of its vertices have neighbors in all the other color classes, one can eliminate color class C assigning to each of its elements a missing color in its neighborhood.

Let a b-vertex be a vertex with neighbors in all other colorings. Then, one can repeat the above procedure until a coloring is obtained where every color class contains a b-vertex, in which case none of the color classes can be eliminated with the same idea. So, one can define a b-coloring as a proper coloring where each color class has a b-vertex.

In the worst case, after successive applications of the above procedure, one get a proper coloring that uses a number of colors equal to the the b-chromatic number of G (denoted $\chi_b(G)$): the maximum k such that G admits a b-coloring with k colors.

A useful upper bound for calculating the b-chromatic number is the following. If G admits a b-coloring with k colors, then there are k vertices of degree at least $k-1$ (the b-vertices of each color class). So, if we set $m(G) = \max\{k|\text{there are } k \text{ vertices of degree at least } k-1\}$, we have that $\chi_b(G) \leq m(G)$.

Note: This method computes a b-coloring that uses at *MOST* k colors. If this method returns a value equal to k, it can not be assumed that k is equal to $\chi_b(G)$. Meanwhile, if it returns any value $k' < k$, this is a certificate that the Grundy number of the given graph is k'.

As $\chi_b(G) \leq m(G)$, it can be assumed that $\chi_b(G) = k$ if `b_coloring(g, k)` returns k when $k = m(G)$.

INPUT:

- `k` – integer; maximum number of colors
- `value_only` – boolean (default: `True`); when set to `True`, only the number of colors is returned. Otherwise, the pair `(nb_colors, coloring)` is returned, where `coloring` is a dictionary associating its color (integer) to each vertex of the graph.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

ALGORITHM:

Integer Linear Program.

EXAMPLES:

The b-chromatic number of a P_5 is equal to 3:

```
sage: from sage.graphs.graph_coloring import b_coloring
sage: g = graphs.PathGraph(5)
sage: b_coloring(g, 5)
3
```

The b-chromatic number of the Petersen Graph is equal to 3:

```
sage: g = graphs.PetersenGraph()
sage: b_coloring(g, 5)
3
```

It would have been sufficient to set the value of `k` to 4 in this case, as $4 = m(G)$.

sage.graphs.graph_coloring.**chromatic_number**(*G*)

Return the chromatic number of the graph.

The chromatic number is the minimal number of colors needed to color the vertices of the graph G.

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import chromatic_number
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: chromatic_number(G)
3

sage: G = graphs.PetersenGraph()
sage: G.chromatic_number()
3
```

sage.graphs.graph_coloring.**edge_coloring**(*g*, *value_only=False*, *vizing=False*, *hex_colors=False*, *solver=None*, *verbose=0*)

Compute chromatic index and edge colorings.

INPUT:

- `g` – a graph.
- `value_only` – boolean (default: `False`):
  - When set to `True`, only the chromatic index is returned
  - When set to `False`, a partition of the edge set into matchings is returned if possible
- `vizing` – boolean (default: `False`):
  - When set to `True`, tries to find a $\Delta + 1$-edge-coloring, where Δ is equal to the maximum degree in the graph
  - When set to `False`, tries to find a Δ-edge-coloring, where Δ is equal to the maximum degree in the graph. If impossible, tries to find and returns a $\Delta + 1$-edge-coloring. This implies that `value_only=False`
- `hex_colors` – boolean (default: `False`); when set to `True`, the partition returned is a dictionary whose keys are colors and whose values are the color classes (ideal for plotting)
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

In the following, Δ is equal to the maximum degree in the graph `g`.

- If `vizing=True` and `value_only=False`, return a partition of the edge set into $\Delta + 1$ matchings.
- If `vizing=False` and `value_only=True`, return the chromatic index.
- If `vizing=False` and `value_only=False`, return a partition of the edge set into the minimum number of matchings.
- If `vizing=True` and `value_only=True`, should return something, but mainly you are just trying to compute the maximum degree of the graph, and this is not the easiest way. By Vizing's theorem, a graph has a chromatic index equal to Δ or to $\Delta + 1$.

Note: In a few cases, it is possible to find very quickly the chromatic index of a graph, while it remains a tedious job to compute a corresponding coloring. For this reason, `value_only = True` can sometimes be much faster, and it is a bad idea to compute the whole coloring if you do not need it !

See also:

- Wikipedia article Edge_coloring for further details on edge coloring
- *chromatic_index()*
- *fractional_chromatic_index()*
- *chromatic_number()*
- *sage.graphs.graph_coloring.vertex_coloring()*

EXAMPLES:

The Petersen graph has chromatic index 4:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: g = graphs.PetersenGraph()
sage: edge_coloring(g, value_only=True, solver='GLPK')
4
sage: edge_coloring(g, value_only=False, solver='GLPK')
[[(0, 1), (2, 3), (4, 9), (5, 7), (6, 8)],
 [(0, 4), (1, 2), (3, 8), (6, 9)],
 [(0, 5), (2, 7)],
 [(1, 6), (3, 4), (5, 8), (7, 9)]]
sage: edge_coloring(g, value_only=False, hex_colors=True, solver='GLPK')
{'#00ffff': [(0, 5), (2, 7)],
 '#7f00ff': [(1, 6), (3, 4), (5, 8), (7, 9)],
 '#7fff00': [(0, 4), (1, 2), (3, 8), (6, 9)],
 '#ff0000': [(0, 1), (2, 3), (4, 9), (5, 7), (6, 8)]}
```

Complete graphs are colored using the linear-time round-robin coloring:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: len(edge_coloring(graphs.CompleteGraph(20)))
19
```

The chromatic index of a non connected graph is the maximum over its connected components:

```
sage: g = graphs.CompleteGraph(4) + graphs.CompleteGraph(10)
sage: edge_coloring(g, value_only=True)
9
```

sage.graphs.graph_coloring.**first_coloring**(*G*, *n=0*, *hex_colors=False*)

Return the first vertex coloring found.

If a natural number n is provided, returns the first found coloring with at least n colors.

INPUT:

- `n` – integer (default: 0); the minimal number of colors to try
- `hex_colors` – boolean (default: `False`); when set to `True`, the partition returned is a dictionary whose keys are colors and whose values are the color classes (ideal for plotting)

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import first_coloring
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: sorted(first_coloring(G, 3))
[[0], [1, 3], [2]]
```

sage.graphs.graph_coloring.**grundy_coloring**(*g*, *k*, *value_only=True*, *solver=None*, *verbose=0*)

Compute Grundy numbers and Grundy colorings.

The method computes the worst-case of a first-fit coloring with less than k colors.

Definition:

A first-fit coloring is obtained by sequentially coloring the vertices of a graph, assigning them the smallest color not already assigned to one of its neighbors. The result is clearly a proper coloring, which usually requires much more colors than an optimal vertex coloring of the graph, and heavily depends on the ordering of the vertices.

The number of colors required by the worst-case application of this algorithm on a graph G is called the Grundy number, written $\Gamma(G)$.

Equivalent formulation:

Equivalently, a Grundy coloring is a proper vertex coloring such that any vertex colored with i has, for every $j < i$, a neighbor colored with j. This can define a Linear Program, which is used here to compute the Grundy number of a graph.

Note: This method computes a grundy coloring using at *MOST* k colors. If this method returns a value equal to k, it can not be assumed that k is equal to $\Gamma(G)$. Meanwhile, if it returns any value $k' < k$, this is a certificate that the Grundy number of the given graph is k'.

As $\Gamma(G) \leq \Delta(G) + 1$, it can also be assumed that $\Gamma(G) = k$ if `grundy_coloring(g, k)` returns k when $k = \Delta(G) + 1$.

INPUT:

- `k` – integer; maximum number of colors
- `value_only` – boolean (default: `True`); when set to `True`, only the number of colors is returned. Otherwise, the pair `(nb_colors, coloring)` is returned, where `coloring` is a dictionary associating its color (integer) to each vertex of the graph.
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

ALGORITHM:

Integer Linear Program.

EXAMPLES:

The Grundy number of a P_4 is equal to 3:

```
sage: from sage.graphs.graph_coloring import grundy_coloring
sage: g = graphs.PathGraph(4)
sage: grundy_coloring(g, 4)
3
```

The Grundy number of the PetersenGraph is equal to 4:

```
sage: g = graphs.PetersenGraph()
sage: grundy_coloring(g, 5)
4
```

It would have been sufficient to set the value of `k` to 4 in this case, as $4 = \Delta(G) + 1$.

`sage.graphs.graph_coloring.`**`linear_arboricity`**(*g*, *plus_one=None*, *hex_colors=False*, *value_only=False*, *solver=None*, *verbose=0*)

Compute the linear arboricity of the given graph.

The linear arboricity of a graph G is the least number $la(G)$ such that the edges of G can be partitioned into linear forests (i.e. into forests of paths).

Obviously, $la(G) \geq \left\lceil \frac{\Delta(G)}{2} \right\rceil$.

It is conjectured in [Aki1980] that $la(G) \leq \left\lceil \frac{\Delta(G)+1}{2} \right\rceil$.

INPUT:

- `plus_one` – integer (default: `None`); whether to use $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ or $\left\lceil \frac{\Delta(G)+1}{2} \right\rceil$ colors.
  - If `0`, computes a decomposition of G into $\left\lceil \frac{\Delta(G)}{2} \right\rceil$ forests of paths
  - If `1`, computes a decomposition of G into $\left\lceil \frac{\Delta(G)+1}{2} \right\rceil$ colors, which is the conjectured general bound.
  - If `plus_one = None` (default), computes a decomposition using the least possible number of colors.
- `hex_colors` – boolean (default: `False`):
  - If `hex_colors = True`, the function returns a dictionary associating to each color a list of edges (meant as an argument to the `edge_colors` keyword of the `plot` method).
  - If `hex_colors = False` (default value), returns a list of graphs corresponding to each color class.
- `value_only` – boolean (default: `False`):
  - If `value_only = True`, only returns the linear arboricity as an integer value.
  - If `value_only = False`, returns the color classes according to the value of `hex_colors`
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.

ALGORITHM:

Linear Programming

COMPLEXITY:

NP-Hard

EXAMPLES:

Obviously, a square grid has a linear arboricity of 2, as the set of horizontal lines and the set of vertical lines are an admissible partition:

```
sage: from sage.graphs.graph_coloring import linear_arboricity
sage: g = graphs.Grid2dGraph(4, 4)
sage: g1,g2 = linear_arboricity(g)
```

Each graph is of course a forest:

```
sage: g1.is_forest() and g2.is_forest()
True
```

Of maximum degree 2:

```
sage: max(g1.degree()) <= 2 and max(g2.degree()) <= 2
True
```

Which constitutes a partition of the whole edge set:

```
sage: all((g1.has_edge(e) or g2.has_edge(e)) for e in g.edge_
↪iterator(labels=None))
True
```

sage.graphs.graph_coloring.**number_of_n_colorings**(G, n)

Compute the number of n-colorings of a graph

INPUT:

- G – a graph
- n – a positive integer; the number of colors

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import number_of_n_colorings
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: number_of_n_colorings(G, 3)
12
```

sage.graphs.graph_coloring.**numbers_of_colorings**(G)

Compute the number of colorings of a graph.

Return the number of n-colorings of the graph G for all n from 0 to $|V|$.

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import numbers_of_colorings
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: numbers_of_colorings(G)
[0, 0, 0, 12, 72]
```

sage.graphs.graph_coloring.**round_robin**(n)

Compute a round-robin coloring of the complete graph on n vertices.

A round-robin coloring of the complete graph G on $2n$ vertices ($V = [0, \ldots, 2n-1]$) is a proper coloring of its edges such that the edges with color i are all the $(i+j, i-j)$ plus the edge $(2n-1, i)$.

If n is odd, one obtain a round-robin coloring of the complete graph through the round-robin coloring of the graph with $n+1$ vertices.

INPUT:

- n – the number of vertices in the complete graph

OUTPUT:

- A *CompleteGraph()* with labelled edges such that the label of each edge is its color.

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import round_robin
sage: round_robin(3).edges()
[(0, 1, 2), (0, 2, 1), (1, 2, 0)]
```

```
sage: round_robin(4).edges()
[(0, 1, 2), (0, 2, 1), (0, 3, 0), (1, 2, 0), (1, 3, 1), (2, 3, 2)]
```

For higher orders, the coloring is still proper and uses the expected number of colors:

```
sage: g = round_robin(9)
sage: sum(Set(e[2] for e in g.edges_incident(v)).cardinality() for v in g) == 2 *␣
↪g.size()
True
sage: Set(e[2] for e in g.edge_iterator()).cardinality()
9
```

```
sage: g = round_robin(10)
sage: sum(Set(e[2] for e in g.edges_incident(v)).cardinality() for v in g) == 2 *␣
↪g.size()
True
sage: Set(e[2] for e in g.edge_iterator()).cardinality()
9
```

sage.graphs.graph_coloring.**vertex_coloring**(*g*, *k=None*, *value_only=False*, *hex_colors=False*, *solver=None*, *verbose=0*)

Compute Vertex colorings and chromatic numbers.

This function can compute the chromatic number of the given graph or test its k-colorability.

See the Wikipedia article Graph_coloring for further details on graph coloring.

INPUT:

- `g` – a graph.
- `k` – integer (default: `None`); tests whether the graph is k-colorable. The function returns a partition of the vertex set in k independent sets if possible and `False` otherwise.
- `value_only` – boolean (default: `False`):
  - When set to `True`, only the chromatic number is returned.
  - When set to `False` (default), a partition of the vertex set into independent sets is returned if possible.
- `hex_colors` – boolean (default: `False`); when set to `True`, the partition returned is a dictionary whose keys are colors and whose values are the color classes (ideal for plotting).
- `solver` – (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

- If `k=None` and `value_only=False`, then return a partition of the vertex set into the minimum possible of independent sets.
- If `k=None` and `value_only=True`, return the chromatic number.
- If `k` is set and `value_only=None`, return `False` if the graph is not k-colorable, and a partition of the vertex set into k independent sets otherwise.
- If `k` is set and `value_only=True`, test whether the graph is k-colorable, and return `True` or `False` accordingly.

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import vertex_coloring
sage: g = graphs.PetersenGraph()
sage: vertex_coloring(g, value_only=True)
3
```

5.2 Interface with Cliquer (clique-related problems)

This module defines functions based on Cliquer, an exact branch-and-bound algorithm developed by Patric R. J. Ostergard and written by Sampo Niskanen.

AUTHORS:

- Nathann Cohen (2009-08-14): Initial version
- Jeroen Demeyer (2011-05-06): Make cliquer interruptible (trac ticket #11252)
- Nico Van Cleemput (2013-05-27): Handle the empty graph (trac ticket #14525)

REFERENCE:

[NO2003]

5.2.1 Methods

`sage.graphs.cliquer.`**`all_cliques`**(*graph*, *min_size=0*, *max_size=0*)

Iterator over the cliques in `graph`.

A clique is an induced complete subgraph. This method is an iterator over all the cliques with size in between `min_size` and `max_size`. By default, this method returns only maximum cliques. Each yielded clique is represented by a list of vertices.

Note: Currently only implemented for undirected graphs. Use *`to_undirected()`* to convert a digraph to an undirected graph.

INPUT:

- `min_size` – integer (default: 0); minimum size of reported cliques. When set to 0 (default), this method searches for maximum cliques. In such case, parameter `max_size` must also be set to 0.
- `max_size` – integer (default: 0); maximum size of reported cliques. When set to 0 (default), the maximum size of the cliques is unbounded. When `min_size` is set to 0, this parameter must be set to 0.

ALGORITHM:

This function is based on Cliquer [NO2003].

EXAMPLES:

```
sage: G = graphs.CompleteGraph(5)
sage: list(sage.graphs.cliquer.all_cliques(G))
[[0, 1, 2, 3, 4]]
sage: list(sage.graphs.cliquer.all_cliques(G, 2, 3))
[[3, 4],
 [2, 3],
 [2, 3, 4],
 [2, 4],
 [1, 2],
 [1, 2, 3],
 [1, 2, 4],
 [1, 3],
 [1, 3, 4],
 [1, 4],
 [0, 1],
```

(continues on next page)

(continued from previous page)

```
 [0, 1, 2],
 [0, 1, 3],
 [0, 1, 4],
 [0, 2],
 [0, 2, 3],
 [0, 2, 4],
 [0, 3],
 [0, 3, 4],
 [0, 4]]
sage: G.delete_edge([1,3])
sage: list(sage.graphs.cliquer.all_cliques(G))
[[0, 2, 3, 4], [0, 1, 2, 4]]
```

Todo: Use the re-entrant functionality of Cliquer [NO2003] to avoid storing all cliques.

sage.graphs.cliquer.**all_max_clique**(*graph*)

Returns the vertex sets of *ALL* the maximum complete subgraphs.

Returns the list of all maximum cliques, with each clique represented by a list of vertices. A clique is an induced complete subgraph, and a maximum clique is one of maximal order.

Note: Currently only implemented for undirected graphs. Use *to_undirected()* to convert a digraph to an undirected graph.

ALGORITHM:

This function is based on Cliquer [NO2003].

EXAMPLES:

```
sage: graphs.ChvatalGraph().cliques_maximum() # indirect doctest
[[0, 1], [0, 4], [0, 6], [0, 9], [1, 2], [1, 5], [1, 7], [2, 3],
 [2, 6], [2, 8], [3, 4], [3, 7], [3, 9], [4, 5], [4, 8], [5, 10],
 [5, 11], [6, 10], [6, 11], [7, 8], [7, 11], [8, 10], [9, 10], [9, 11]]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_maximum()
[[0, 1, 2], [0, 1, 3]]
sage: C = graphs.PetersenGraph()
sage: C.cliques_maximum()
[[0, 1], [0, 4], [0, 5], [1, 2], [1, 6], [2, 3], [2, 7], [3, 4],
 [3, 8], [4, 9], [5, 7], [5, 8], [6, 8], [6, 9], [7, 9]]
sage: C = Graph('DJ{')
sage: C.cliques_maximum()
[[1, 2, 3, 4]]
```

sage.graphs.cliquer.**clique_number**(*graph*)

Returns the size of the largest clique of the graph (clique number).

Note: Currently only implemented for undirected graphs. Use *to_undirected()* to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.clique_number()
4
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.clique_number()
3
```

sage.graphs.cliquer.**max_clique**(*graph*)

Returns the vertex set of a maximum complete subgraph.

Note: Currently only implemented for undirected graphs. Use *to_undirected()* to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: C = graphs.PetersenGraph()
sage: from sage.graphs.cliquer import max_clique
sage: max_clique(C)
[7, 9]
```

5.3 Centrality

This module is meant for all functions related to centrality in networks.

centrality_betweenness()	Return the centrality betweenness of G
centrality_closeness_top_k()	Return the k most closeness central vertices of G
centrality_closeness_random_k()	Return an estimation of the closeness centrality of G.

5.3.1 Functions

sage.graphs.centrality.**centrality_betweenness**(*G*, *exact=False*, *normalize=True*)

Return the centrality betweenness of G

The centrality betweenness of a vertex $v \in G$ is defined by:

$$c(v) = \sum_{s \neq v \neq t} \frac{\#\{\text{shortest } st - \text{paths containing v}\}}{\#\{\text{shortest } st - \text{paths}\}}$$

For more information, see the Wikipedia article Betweenness_centrality.

INPUT:

- `G` – a (di)graph
- `exact` – boolean (default: `False`); whether to compute over rationals or on `double` C variables.
- `normalize` – boolean (default: `True`); whether to renormalize the values by dividing them by $\binom{n-1}{2}$ (for graphs) or $2\binom{n-1}{2}$ (for digraphs).

ALGORITHM:

To compute $c(v)$, we fix s and define $c_s(v)$ as the centrality of v *due to* s, obtained from the formula above by running the sum over t only. We obtain $c(v) = \sum_{s \neq v} c_s(v)$.

For every vertex s, we compute the value of $c_s(v)$ for all v, using the following remark (see [Brandes01]):

Let $v_1, \ldots, v_k$ be the out-neighbors of v such that $dist(s, v_i) = dist(s, v) + 1$. Then

$$c_s(v) = \sum_{1 \leq i \leq k} c_s(v_i) \frac{\#\{\text{shortest } sv_i - \text{paths}\}}{\#\{\text{shortest } sv - \text{paths}\}}$$

The number of shortest paths between s and every other vertex can be computed with a slightly modified BFS. While running this BFS we can also store the list of the vertices $v_1, \ldots, v_k$ associated with each v.

EXAMPLES:

```
sage: from sage.graphs.centrality import centrality_betweenness
sage: centrality_betweenness(digraphs.Circuit(6)) # abs tol 1e-10
{0: 0.5, 1: 0.5, 2: 0.5, 3: 0.5, 4: 0.5, 5: 0.5}
sage: centrality_betweenness(graphs.CycleGraph(6)) # abs tol 1e-10
{0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2, 4: 0.2, 5: 0.2}
```

Exact computations:

```
sage: graphs.PetersenGraph().centrality_betweenness(exact=True)
{0: 1/12, 1: 1/12, 2: 1/12, 3: 1/12, 4: 1/12, 5: 1/12, 6: 1/12, 7: 1/12, 8: 1/12,␣
↪9: 1/12}
```

sage.graphs.centrality.**centrality_closeness_random_k**(*G*, *k=1*)

Return an estimation of the closeness centrality of G.

The algorithm first randomly selects a set S of k vertices. Then it computes shortest path distances from each vertex in S (using Dijkstra for weighted graph and breadth-first-search (BFS) for unweighted graph) and uses this knowledge to estimate the closeness centrality of all vertices.

For more information, see [EDI2014].

INPUT:

- `G` – an undirected connected Graph
- `k` – integer (default: 1); number of random nodes to choose

OUTPUT:

A dictionary associating to each vertex its estimated closeness centrality.

EXAMPLES:

Estimation of the closeness centrality of the Petersen Graph when $k == n$:

```
sage: from sage.graphs.centrality import centrality_closeness_random_k
sage: G = graphs.PetersenGraph()
sage: centrality_closeness_random_k(G, 10)
{0: 0.6,
 1: 0.6,
 2: 0.6,
 3: 0.6,
 4: 0.6,
 5: 0.6,
 6: 0.6,
 7: 0.6,
 8: 0.6,
 9: 0.6}
```

sage.graphs.centrality.**centrality_closeness_top_k**(*G*, *k=1*, *verbose=0*)

Compute the k vertices with largest closeness centrality.

The algorithm is based on performing a breadth-first-search (BFS) from each vertex, and to use bounds in order to cut these BFSes as soon as possible. If k is small, it is much more efficient than computing all centralities with *centrality_closeness()*. Conversely, if k is close to the number of nodes, the running-time is approximately the same (it might even be a bit longer, because more computations are needed).

For more information, see [BCM15]. The algorithm does not work on weighted graphs.

INPUT:

- G – a Sage Graph or DiGraph;
- k – integer (default: 1); the algorithm will return the k vertices with largest closeness centrality. This value should be between 1 and the number of vertices with positive (out)degree, because the closeness centrality is not defined for vertices with (out)degree 0. If k is bigger than this value, the output will contain all vertices of positive (out)degree.
- verbose – integer (default: 0); define how "verbose" the algorithm should be. If 0, nothing is printed, if 1, we print only the performance ratio at the end of the algorithm, if 2, we print partial results every 1000 visits, if 3, we print partial results after every visit.

OUTPUT:

An ordered list of k pairs $(closv, v)$, where v is one of the k most central vertices, and $closv$ is its closeness centrality. If k is bigger than the number of vertices with positive (out)degree, the list might be smaller.

EXAMPLES:

```
sage: from sage.graphs.centrality import centrality_closeness_top_k
sage: g = graphs.PathGraph(10)
sage: centrality_closeness_top_k(g, 4, 1)
Final performance ratio: 0.711111111111...
[(0.36, 5),
 (0.36, 4),
 (0.3333333333333333, 6),
 (0.3333333333333333, 3)]
sage: g = digraphs.Path(10)
sage: centrality_closeness_top_k(g, 5, 1)
Final performance ratio: 0.422222222222...
[(0.2, 0),
 (0.19753086419753085, 1),
 (0.1944444444444442, 2),
 (0.19047619047619047, 3),
 (0.18518518518518517, 4)]
```

5.4 Asteroidal triples

This module contains the following function:

is_asteroidal_triple_free()	Test if the input graph is asteroidal triple-free

5.4.1 Definition

Three independent vertices of a graph form an *asteroidal triple* if every two of them are connected by a path avoiding the neighborhood of the third one. A graph is *asteroidal triple-free* (*AT-free*, for short) if it contains no asteroidal triple

[LB1962].

Use `graph_classes.AT_free.description()` to get some known properties of AT-free graphs, or visit this page.

5.4.2 Algorithm

This module implements the *Straightforward algorithm* recalled in [Koh2004] and due to [LB1962] for testing if a graph is AT-free or not. This algorithm has time complexity in $O(n^3)$ and space complexity in $O(n^2)$.

This algorithm uses the *connected structure* of the graph, stored into a $n \times n$ matrix M. This matrix is such that $M[u][v] == 0$ if $v \in (\{u\} \cup N(u))$, and otherwise $M[u][v]$ is the unique identifier (a strictly positive integer) of the connected component of $G \setminus (\{u\} \cup N(u))$ to which v belongs. This connected structure can be computed in time $O(n(n+m))$ using n BFS.

Now, a triple $u, v, w \in V$ is an asteroidal triple if and only if it satisfies $M[u][v] == M[u][w]$ and $M[v][u] == M[v][w]$ and $M[w][u] == M[w][v]$, assuming all these values are positive. Indeed, if $M[u][v] == M[u][w]$, v and w are in the same connected component of $G \setminus (\{u\} \cup N(u))$, and so there is a path between v and w avoiding the neighborhood of u. The algorithm iterates over all triples.

5.4.3 Functions

`sage.graphs.asteroidal_triples.`**`is_asteroidal_triple_free`**(*G*, *certificate=False*)

Test if the input graph is asteroidal triple-free

An independent set of three vertices such that each pair is joined by a path that avoids the neighborhood of the third one is called an *asteroidal triple*. A graph is asteroidal triple-free (AT-free) if it contains no asteroidal triples. See the *module's documentation* for more details.

This method returns `True` is the graph is AT-free and `False` otherwise.

INPUT:

- `G` – a Graph
- `certificate` – boolean (default: `False`); by default, this method returns `True` if the graph is asteroidal triple-free and `False` otherwise. When `certificate==True`, this method returns in addition a list of three vertices forming an asteroidal triple if such a triple is found, and the empty list otherwise.

EXAMPLES:

The complete graph is AT-free, as well as its line graph:

```
sage: G = graphs.CompleteGraph(5)
sage: G.is_asteroidal_triple_free()
True
sage: G.is_asteroidal_triple_free(certificate=True)
(True, [])
sage: LG = G.line_graph()
sage: LG.is_asteroidal_triple_free()
True
sage: LLG = LG.line_graph()
sage: LLG.is_asteroidal_triple_free()
False
```

The PetersenGraph is not AT-free:

```
sage: from sage.graphs.asteroidal_triples import *
sage: G = graphs.PetersenGraph()
sage: G.is_asteroidal_triple_free()
False
sage: G.is_asteroidal_triple_free(certificate=True)
(False, [0, 2, 6])
```

5.5 Independent sets

This module implements the *IndependentSets* class which can be used to :

- List the independent sets (or cliques) of a graph
- Count them (which is obviously faster)
- Test whether a set of vertices is an independent set

It can also be restricted to focus on (inclusionwise) maximal independent sets. See the documentation of *IndependentSets* for actual examples.

5.5.1 Classes and methods

class sage.graphs.independent_sets.**IndependentSets**

Bases: object

The set of independent sets of a graph.

For more information on independent sets, see the Wikipedia article Independent_set_(graph_theory).

INPUT:

- G – a graph
- maximal – boolean (default: False); whether to only consider (inclusionwise) maximal independent sets.
- complement – boolean (default: False); whether to consider the graph's complement (i.e. cliques instead of independent sets).

ALGORITHM:

The enumeration of independent sets is done naively : given an independent set, this implementation considers all ways to add a new vertex to it (while keeping it an independent set), and then creates new independent sets from all those that were created this way.

The implementation, however, is not recursive.

Note: This implementation of the enumeration of *maximal* independent sets is not much faster than NetworkX', which is surprising as it is written in Cython. This being said, the algorithm from NetworkX appears to be sligthly different from this one, and that would be a good thing to explore if one wants to improve the implementation.

A simple generalization can also be done without too much modifications: iteration through independent sets with given size bounds (minimum and maximum number of vertices allowed).

EXAMPLES:

Listing all independent sets of the Claw graph:

```
sage: from sage.graphs.independent_sets import IndependentSets
sage: g = graphs.ClawGraph()
sage: I = IndependentSets(g)
sage: list(I)
[[0], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3], []]
```

Count them:

```
sage: I.cardinality()
9
```

List only the maximal independent sets:

```
sage: Im = IndependentSets(g, maximal=True)
sage: list(Im)
[[0], [1, 2, 3]]
```

And count them:

```
sage: Im.cardinality()
2
```

One can easily count the number of independent sets of each cardinality:

```
sage: g = graphs.PetersenGraph()
sage: number_of = [0] * g.order()
sage: for x in IndependentSets(g):
....:     number_of[len(x)] += 1
sage: number_of
[1, 10, 30, 30, 5, 0, 0, 0, 0, 0]
```

It is also possible to define an iterator over all independent sets of a given cardinality. Note, however, that Sage will generate them *all*, to return only those that satisfy the cardinality constraints. Getting the list of independent sets of size 4 in this way can thus take a very long time:

```
sage: is4 = (x for x in IndependentSets(g) if len(x) == 4)
sage: list(is4)
[[0, 2, 8, 9], [0, 3, 6, 7], [1, 3, 5, 9], [1, 4, 7, 8], [2, 4, 5, 6]]
```

Given a subset of the vertices, it is possible to test whether it is an independent set:

```
sage: g = graphs.DurerGraph()
sage: I = IndependentSets(g)
sage: [0, 2] in I
True
sage: [0, 3, 5] in I
False
```

If an element of the subset is not a vertex, then an error is raised:

```
sage: [0, 'a', 'b', 'c'] in I
Traceback (most recent call last):
...
ValueError: a is not a vertex of the graph
```

cardinality()

Compute and return the number of independent sets.

5.6 Comparability and permutation graphs

This module implements method related to Wikipedia article Comparability_graph and Wikipedia article Permutation_graph, that is, for the moment, only recognition algorithms.

Most of the information found here can alo be found in [ST1994] or [Sha1997].

The following methods are implemented in this module

`is_comparability_MILP()`	Tests whether the graph is a comparability graph (MILP)
`greedy_is_comparability()`	Tests whether the graph is a comparability graph (greedy algorithm)
`greedy_is_comparability_certificate()`	Tests whether the graph is a comparability graph and returns certificates (greedy algorithm)
`is_comparability()`	Tests whether the graph is a comparability graph
`is_permutation()`	Tests whether the graph is a permutation graph.
`is_transitive()`	Tests whether the digraph is transitive.

Author:

- Nathann Cohen 2012-04

5.6.1 Graph classes

Comparability graphs

A graph is a comparability graph if it can be obtained from a poset by adding an edge between any two elements that are comparable. Co-comparability graph are complements of such graphs, i.e. graphs built from a poset by adding an edge between any two incomparable elements.

For more information on comparability graphs, see the Wikipedia article Comparability_graph.

Permutation graphs

Definitions:

- A permutation $\pi = \pi_1\pi_2\dots\pi_n$ defines a graph on n vertices such that $i \sim j$ when π reverses i and j (i.e. when $i < j$ and $\pi_j < \pi_i$. A graph is a permutation graph whenever it can be built through this construction.
- A graph is a permutation graph if it can be build from two parallel lines are the intersection graph of segments intersecting both lines.
- A graph is a permutation graph if it is both a comparability graph and a co-comparability graph.

For more information on permutation graphs, see the Wikipedia article Permutation_graph.

5.6.2 Recognition algorithm for comparability graphs

Greedy algorithm

This algorithm attempts to build a transitive orientation of a given graph G, that is an orientation D such that for any directed uv-path of D there exists in D an edge uv. This already determines a notion of equivalence between some edges of G :

> In G, two edges uv and uv' (incident to a common vertex u) such that $vv' \notin G$ need necessarily be oriented *the same way* (that is that they should either both *leave* or both *enter* u). Indeed, if one enters G while the other leaves it, these two edges form a path of length two, which is not possible in any transitive orientation of G as $vv' \notin G$.

Hence, we can say that in this case a *directed edge* uv is equivalent to a *directed edge* uv' (to mean that if one belongs to the transitive orientation, the other one must be present too) in the same way that vu is equivalent to $v'u$. We can thus define equivalence classes on oriented edges, to represent set of edges that imply each other. We can thus define C_{uv}^G to be the equivalence class in G of the oriented edge uv.

Of course, if there exists a transitive orientation of a graph G, then no edge uv implies its contrary vu, i.e. it is necessary to ensure that $\forall uv \in G, vu \notin C_{uv}^G$. The key result on which the greedy algorithm is built is the following (see [ST1994]):

Theorem – The following statements are equivalent :

- G is a comparability graph
- $\forall uv \in G, vu \notin C_{uv}^G$
- The edges of G can be partitionned into $B_1, ..., B_k$ where B_i is the equivalence class of some oriented edge in $G - B_1 - \cdots - B_{i-1}$

Hence, ensuring that a graph is a comparability graph can be done by checking that no equivalence class is contradictory. Building the orientation, however, requires to build equivalence classes step by step until an orientation has been found for all of them.

Mixed Integer Linear Program

A MILP formulation is available to check the other methods for correction. It is easily built :

To each edge are associated two binary variables (one for each possible direction). We then ensure that each triangle is transitively oriented, and that each pair of incident edges uv, uv' such that $vv' \notin G$ do not create a 2-path.

Here is the formulation:

Maximize : Nothing

Such that :

$\forall uv \in G$

$\cdot\ o_{uv} + o_{vu} = 1$

$\forall u \in G, \forall v, v' \in N(v)$ such that $vv' \notin G$

$\cdot\ o_{uv} + o_{v'u} - o_{v'v} \leq 1$

$\cdot\ o_{uv'} + o_{vu} - o_{vv'} \leq 1$

$\forall u \in G, \forall v, v' \in N(v)$ such that $vv' \in G$

$\cdot\ o_{uv} + o_{v'u} \leq 1$

$\cdot\ o_{uv'} + o_{vu} \leq 1$

o_{uv} is a binary variable

Note: The MILP formulation is usually much slower than the greedy algorithm. This MILP has been implemented to check the results of the greedy algorithm that has been implemented to check the results of a faster algorithm which has not been implemented yet.

5.6.3 Certificates

Comparability graphs

The *yes*-certificates that a graph is a comparability graphs are transitive orientations of it. The *no*-certificates, on the other hand, are odd cycles of such graph. These odd cycles have the property that around each vertex v of the cycle its two incident edges must have the same orientation (toward v, or outward v) in any transitive orientation of the graph.

This is impossible whenever the cycle has odd length. Explanations are given in the "Greedy algorithm" part of the previous section.

Permutation graphs

Permutation graphs are precisely the intersection of comparability graphs and co-comparability graphs. Hence, negative certificates are precisely negative certificates of comparability or co-comparability. Positive certificates are a pair of permutations that can be used through *PermutationGraph()* (whose documentation says more about what these permutations represent).

5.6.4 Implementation details

Test that the equivalence classes are not self-contradictory

This is done by a call to `Graph.is_bipartite()`, and here is how :

> Around a vertex u, any two edges uv, uv' such that $vv' \notin G$ are equivalent. Hence, the equivalence classe of edges around a vertex are precisely the connected components of the complement of the graph induced by the neighbors of u.
>
> In each equivalence class (around a given vertex u), the edges should all have the same orientation, i.e. all should go toward u at the same time, or leave it at the same time. To represent this, we create a graph with vertices for all equivalent classes around all vertices of G, and link (v, C) to (u, C') if $u \in C$ and $v \in C'$.
>
> A bipartite coloring of this graph with colors 0 and 1 tells us that the edges of an equivalence class C around u should be directed toward u if (u, C) is colored with 0, and outward if (u, C) is colored with 1.
>
> If the graph is not bipartite, this is the proof that some equivalence class is self-contradictory !

Note: The greedy algorithm implemented here is just there to check the correction of more complicated ones, and it is reaaaaaaaaaaaalllly bad whenever you look at it with performance in mind.

5.6.5 Methods

`sage.graphs.comparability.`**`greedy_is_comparability`**(*g*, *no_certificate=False*, *equivalence_class=False*)

Tests whether the graph is a comparability graph (greedy algorithm)

This method only returns no-certificates.

To understand how this method works, please consult the documentation of the *comparability module*.

INPUT:

- `no_certificate` – whether to return a *no*-certificate when the graph is not a comparability graph. This certificate is an odd cycle of edges, each of which implies the next. It is set to `False` by default.
- `equivalence_class` – whether to return an equivalence class if the graph is a comparability graph.

OUTPUT:

- If the graph is a comparability graph and `no_certificate = False`, this method returns `True` or `(True, an_equivalence_class)` according to the value of `equivalence_class`.
- If the graph is *not* a comparability graph, this method returns `False` or `(False, odd_cycle)` according to the value of `no_certificate`.

EXAMPLES:

The Petersen Graph is not transitively orientable:

```
sage: from sage.graphs.comparability import greedy_is_comparability as is_
↪comparability
sage: g = graphs.PetersenGraph()
sage: is_comparability(g)
False
sage: is_comparability(g, no_certificate=True)  # py2
(False, [0, 4, 9, 6, 1, 0])
sage: is_comparability(g, no_certificate=True)  # py3
(False, [2, 1, 0, 4, 3, 2])
```

But the Bull graph is:

```
sage: g = graphs.BullGraph()
sage: is_comparability(g)
True
```

sage.graphs.comparability.**greedy_is_comparability_with_certificate**(*g*, *certificate=False*)

Tests whether the graph is a comparability graph and returns certificates(greedy algorithm).

This method can return certificates of both *yes* and *no* answers.

To understand how this method works, please consult the documentation of the *comparability module*.

INPUT:

- `certificate` (boolean) – whether to return a certificate. *Yes*-answers the certificate is a transitive orientation of G, and a *no* certificates is an odd cycle of sequentially forcing edges.

EXAMPLES:

The 5-cycle or the Petersen Graph are not transitively orientable:

```
sage: from sage.graphs.comparability import greedy_is_comparability_with_
↪certificate as is_comparability
sage: is_comparability(graphs.CycleGraph(5), certificate=True)  # py2
(False, [1, 2, 3, 4, 0, 1])
sage: is_comparability(graphs.CycleGraph(5), certificate=True)  # py3
(False, [2, 1, 0, 4, 3, 2])
sage: g = graphs.PetersenGraph()
sage: is_comparability(g)
False
sage: is_comparability(g, certificate=True)  # py2
(False, [0, 4, 9, 6, 1, 0])
sage: is_comparability(g, certificate=True)  # py3
(False, [2, 1, 0, 4, 3, 2])
```

But the Bull graph is:

```
sage: g = graphs.BullGraph()
sage: is_comparability(g)
True
sage: is_comparability(g, certificate = True)
(True, Digraph on 5 vertices)
sage: is_comparability(g, certificate = True)[1].is_transitive()
True
```

sage.graphs.comparability.**is_comparability**(*g*, *algorithm='greedy'*, *certificate=False*, *check=True*, *solver=None*, *verbose=0*)

Tests whether the graph is a comparability graph

INPUT:

- `algorithm` – choose the implementation used to do the test.
  - `"greedy"` – a greedy algorithm (see the documentation of the *comparability module*).
  - `"MILP"` – a Mixed Integer Linear Program formulation of the problem. Beware, for this implementation is unable to return negative certificates ! When `certificate = True`, negative certificates are always equal to `None`. True certificates are valid, though.
- `certificate` (boolean) – whether to return a certificate. *Yes*-answers the certificate is a transitive orientation of G, and a *no* certificates is an odd cycle of sequentially forcing edges.
- `check` (boolean) – whether to check that the yes-certificates are indeed transitive. As it is very quick compared to the rest of the operation, it is enabled by default.
- `solver` – (default: `None`); Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: from sage.graphs.comparability import is_comparability
sage: g = graphs.PetersenGraph()
sage: is_comparability(g)
False
sage: is_comparability(graphs.CompleteGraph(5), certificate=True)
(True, Digraph on 5 vertices)
```

sage.graphs.comparability.**is_comparability_MILP**(*g*, *certificate=False*, *solver=None*, *verbose=0*)

Tests whether the graph is a comparability graph (MILP)

INPUT:

- `certificate` (boolean) – whether to return a certificate for yes instances. This method can not return negative certificates.
- `solver` – (default: `None`); Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

The 5-cycle or the Petersen Graph are not transitively orientable:

```
sage: from sage.graphs.comparability import is_comparability_MILP as is_
↪comparability
sage: is_comparability(graphs.CycleGraph(5), certificate = True)
(False, None)
sage: g = graphs.PetersenGraph()
sage: is_comparability(g, certificate = True)
(False, None)
```

But the Bull graph is:

```
sage: g = graphs.BullGraph()
sage: is_comparability(g)
True
sage: is_comparability(g, certificate = True)
(True, Digraph on 5 vertices)
sage: is_comparability(g, certificate = True)[1].is_transitive()
True
```

sage.graphs.comparability.**is_permutation**(*g*, *algorithm='greedy'*, *certificate=False*, *check=True*, *solver=None*, *verbose=0*)

Tests whether the graph is a permutation graph.

For more information on permutation graphs, refer to the documentation of the *comparability module*.

INPUT:

- `algorithm` – choose the implementation used for the subcalls to *is_comparability()*.
  - `"greedy"` – a greedy algorithm (see the documentation of the *comparability module*).
  - `"MILP"` – a Mixed Integer Linear Program formulation of the problem. Beware, for this implementation is unable to return negative certificates ! When `certificate = True`, negative certificates are always equal to `None`. True certificates are valid, though.
- `certificate` (boolean) – whether to return a certificate for the answer given. For `True` answers the certificate is a permutation, for `False` answers it is a no-certificate for the test of comparability or co-comparability.
- `check` (boolean) – whether to check that the permutations returned indeed create the expected Permutation graph. Pretty cheap compared to the rest, hence a good investment. It is enabled by default.
- `solver` – (default: `None`); Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

Note: As the `True` certificate is a `Permutation` object, the segment intersection model of the permutation graph can be visualized through a call to `Permutation.show`.

EXAMPLES:

A permutation realizing the bull graph:

```
sage: from sage.graphs.comparability import is_permutation
sage: g = graphs.BullGraph()
sage: _ , certif = is_permutation(g, certificate=True)
sage: h = graphs.PermutationGraph(*certif)
sage: h.is_isomorphic(g)
True
```

Plotting the realization as an intersection graph of segments:

```
sage: true, perm = is_permutation(g, certificate=True)
sage: p1 = Permutation([nn+1 for nn in perm[0]])
sage: p2 = Permutation([nn+1 for nn in perm[1]])
sage: p = p2 * p1.inverse()
sage: p.show(representation = "braid")
```

sage.graphs.comparability.**is_transitive**(*g*, *certificate=False*)

Tests whether the digraph is transitive.

A digraph is transitive if for any pair of vertices $u, v \in G$ linked by a uv-path the edge uv belongs to G.

INPUT:

- `certificate` – whether to return a certificate for negative answers.
  - If `certificate = False` (default), this method returns `True` or `False` according to the graph.
  - If `certificate = True`, this method either returns `True` answers or yield a pair of vertices uv such that there exists a uv-path in G but $uv \notin G$.

EXAMPLES:

```
sage: digraphs.Circuit(4).is_transitive()
False
sage: digraphs.Circuit(4).is_transitive(certificate=True)
(0, 2)
sage: digraphs.RandomDirectedGNP(30,.2).is_transitive()
False
sage: D = digraphs.DeBruijn(5, 2)
sage: D.is_transitive()
False
sage: cert = D.is_transitive(certificate=True)
sage: D.has_edge(*cert)
False
sage: D.shortest_path(*cert) != []
True
sage: digraphs.RandomDirectedGNP(20,.2).transitive_closure().is_transitive()
True
```

5.7 Line graphs

This module gather everything which is related to line graphs. Right now, this amounts to the following functions :

line_graph()	Computes the line graph of a given graph
is_line_graph()	Check whether a graph is a line graph
root_graph()	Computes the root graph corresponding to the given graph

Author:

- Nathann Cohen (01-2013), *root_graph()* method and module documentation. Written while listening to Nina Simone *"I wish I knew how it would feel to be free"*. Crazy good song. And *"Prendre ta douleur"*, too.
- David Coudert (10-2018), use maximal cliques iterator in *root_graph()*, and use *root_graph()* instead of forbidden subgraph search in *is_line_graph()* (trac ticket #26444).

5.7.1 Definition

Given a graph G, the *line graph* $L(G)$ of G is the graph such that

$$V(L(G)) = E(G)$$
$$E(L(G)) = \{(e, e') : \text{ and } e, e' \text{ have a common endpoint in } G\}$$

The definition is extended to directed graphs. In this situation, there is an arc (e, e') in $L(G)$ if the destination of e is the origin of e'.

For more information, see the Wikipedia article Line_graph.

5.7.2 Root graph

A graph whose line graph is LG is called the *root graph* of LG. The root graph of a (connected) graph is unique ([Whi1932], [Har1969]), except when $LG = K_3$, as both $L(K_3)$ and $L(K_{1,3})$ are equal to K_3.

Here is how we can *"see"* G by staring (very intently) at LG :

> A graph LG is the line graph of G if there exists a collection $(S_v)_{v\in G}$ of subsets of $V(LG)$ such that :
>
> - Every S_v is a complete subgraph of LG.
> - Every $v \in LG$ belongs to exactly two sets of the family $(S_v)_{v\in G}$.
> - Any two sets of $(S_v)_{v\in G}$ have at most one common elements
> - For any edge $(u, v) \in LG$ there exists a set of $(S_v)_{v\in G}$ containing both u and v.
>
> In this family, each set S_v represent a vertex of G, and contains "the set of edges incident to v in G". Two elements $S_v, S_{v'}$ have a nonempty intersection whenever vv' is an edge of G.
>
> Hence, finding the root graph of LG is the job of finding this collection of sets.

In particular, what we know for sure is that a maximal clique S of size 2 or ≥ 4 in LG corresponds to a vertex of degree $|S|$ in G, whose incident edges are the elements of S itself.

The main problem lies with maximal cliques of size 3, i.e. triangles. Those we have to split into two categories, *even* and *odd* triangles :

> A triangle $\{e_1, e_2, e_3\} \subseteq V(LG)$ is said to be an *odd* triangle if there exists a vertex $e \in V(G)$ incident to exactly *one* or *all* of $\{e_1, e_2, e_3\}$, and it is said to be *even* otherwise.

The very good point of this definition is that an inclusionwise maximal clique which is an odd triangle will always correspond to a vertex of degree 3 in G, while an even triangle could result from either a vertex of degree 3 in G or a triangle in G. And in order to build the root graph we obviously have to decide *which*.

Beineke proves in [Bei1970] that the collection of sets we are looking for can be easily found. Indeed it turns out that it is the union of :

1. The family of all maximal cliques of LG of size 2 or ≥ 4, as well as all odd triangles.
2. The family of all pairs of adjacent vertices which appear in exactly *one* maximal clique which is an even triangle.

There are actually four special cases to which the decomposition above does not apply, i.e. graphs containing an edge which belongs to exactly two even triangles. We deal with those independently.

- The `Complete graph` K_3.
- The `Diamond graph` – the line graph of $K_{1,3}$ plus an edge.
- The `Wheel graph` on $4 + 1$ vertices – the line graph of the `Diamond graph`
- The `Octahedron` – the line graph of K_4.

This decomposition turns out to be very easy to implement :-)

Warning: Even though the root graph is *NOT UNIQUE* for the triangle, this method returns $K_{1,3}$ (and not K_3) in this case. Pay *very close* attention to that, for this answer is not theoretically correct : there is no unique answer in this case, and we deal with it by returning one of the two possible answers.

5.7.3 Functions

sage.graphs.line_graph.**is_line_graph**(*g*, *certificate=False*)

Tests wether the graph is a line graph.

INPUT:

- `certificate` (boolean) – whether to return a certificate along with the boolean result. Here is what happens when `certificate = True`:
  - If the graph is not a line graph, the method returns a pair `(b, subgraph)` where `b` is `False` and `subgraph` is a subgraph isomorphic to one of the 9 forbidden induced subgraphs of a line graph.
  - If the graph is a line graph, the method returns a triple `(b,R,isom)` where `b` is `True`, `R` is a graph whose line graph is the graph given as input, and `isom` is a map associating an edge of `R` to each vertex of the graph.

Note: This method wastes a bit of time when the input graph is not connected. If you have performance in mind, it is probably better to only feed it with connected graphs only.

See also:

- The *line_graph* module.
- *line_graph_forbidden_subgraphs()* – the forbidden subgraphs of a line graph.
- *line_graph()*

EXAMPLES:

A complete graph is always the line graph of a star:

```
sage: graphs.CompleteGraph(5).is_line_graph()
True
```

The Petersen Graph not being claw-free, it is not a line graph:

```
sage: graphs.PetersenGraph().is_line_graph()
False
```

This is indeed the subgraph returned:

```
sage: C = graphs.PetersenGraph().is_line_graph(certificate=True)[1]
sage: C.is_isomorphic(graphs.ClawGraph())
True
```

The house graph is a line graph:

```
sage: g = graphs.HouseGraph()
sage: g.is_line_graph()
True
```

But what is the graph whose line graph is the house ?:

```
sage: is_line, R, isom = g.is_line_graph(certificate=True)
sage: R.sparse6_string()
':DaHI~'
sage: R.show()
```

(continues on next page)

(continued from previous page)

```
sage: isom
{0: (0, 1), 1: (0, 2), 2: (1, 3), 3: (2, 3), 4: (3, 4)}
```

sage.graphs.line_graph.**line_graph**(*self*, *labels=True*)

Returns the line graph of the (di)graph.

INPUT:

- `labels` – boolean (default: `True`); whether edge labels should be taken in consideration. If `labels=True`, the vertices of the line graph will be triples `(u,v,label)`, and pairs of vertices otherwise.

The line graph of an undirected graph G is an undirected graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G. In other words, an edge in H represents a path of length 2 in G.

The line graph of a directed graph G is a directed graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G and the terminal vertex of e is the initial vertex of f. In other words, an edge in H represents a (directed) path of length 2 in G.

Note: As a *Graph* object only accepts hashable objects as vertices (and as the vertices of the line graph are the edges of the graph), this code will fail if edge labels are not hashable. You can also set the argument `labels=False` to ignore labels.

See also:

- The *line_graph* module.
- *line_graph_forbidden_subgraphs()* – the forbidden subgraphs of a line graph.
- *is_line_graph()* – tests whether a graph is a line graph.

EXAMPLES:

```
sage: g = graphs.CompleteGraph(4)
sage: h = g.line_graph()
sage: h.vertices()
[(0, 1, None),
(0, 2, None),
(0, 3, None),
(1, 2, None),
(1, 3, None),
(2, 3, None)]
sage: h.am()
[0 1 1 1 1 0]
[1 0 1 1 0 1]
[1 1 0 0 1 1]
[1 1 0 0 1 1]
[1 0 1 1 0 1]
[0 1 1 1 1 0]
sage: h2 = g.line_graph(labels=False)
sage: h2.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: h2.am() == h.am()
True
sage: g = DiGraph([[1..4], lambda i,j: i < j])
```

(continues on next page)

(continued from previous page)

```
sage: h = g.line_graph()
sage: h.vertices()
[(1, 2, None),
(1, 3, None),
(1, 4, None),
(2, 3, None),
(2, 4, None),
(3, 4, None)]
sage: h.edges()
[((1, 2, None), (2, 3, None), None),
 ((1, 2, None), (2, 4, None), None),
 ((1, 3, None), (3, 4, None), None),
 ((2, 3, None), (3, 4, None), None)]
```

sage.graphs.line_graph.**root_graph**(*g*, *verbose=False*)

Computes the root graph corresponding to the given graph

See the documentation of *sage.graphs.line_graph* to know how it works.

INPUT:

- g – a graph
- verbose – boolean (default: False); display some information about what is happening inside of the algorithm.

Note: It is best to use this code through *is_line_graph()*, which first checks that the graph is indeed a line graph, and deals with the disconnected case. But if you are sure of yourself, dig in !

Warning:

- This code assumes that the graph is connected.

5.8 Spanning trees

This module is a collection of algorithms on spanning trees. Also included in the collection are algorithms for minimum spanning trees. See the book [JNC2010] for descriptions of spanning tree algorithms, including minimum spanning trees.

See also:

- *GenericGraph.min_spanning_tree*.

Todo:

- Rewrite *kruskal()* to use priority queues.
- Parallel version of Boruvka's algorithm.
- Randomized spanning tree construction.

5.8.1 Methods

sage.graphs.spanning_tree.**boruvka**(*G*, *wfunction=None*, *check=False*, *by_weight=True*)

Minimum spanning tree using Boruvka's algorithm.

This function assumes that we can only compute minimum spanning trees for undirected graphs. Such graphs can be weighted or unweighted, and they can have multiple edges (since we are computing the minimum spanning tree, only the minimum weight among all (u, v)-edges is considered, for each pair of vertices u, v).

INPUT:

- `G` – an undirected graph.
- `wfunction` – weight function (default: `None`); a function that inputs an edge `e` and outputs its weight. An edge has the form `(u,v,l)`, where `u` and `v` are vertices, `l` is a label (that can be of any kind). The `wfunction` can be used to transform the label into a weight. In particular:
  - if `wfunction` is not `None`, the weight of an edge `e` is `wfunction(e)`;
  - if `wfunction` is `None` (default) and `g` is weighted (that is, `g.weighted()==True`), the weight of an edge `e=(u,v,l)` is `l`, independently on which kind of object `l` is: the ordering of labels relies on Python's operator `<`;
  - if `wfunction` is `None` and `g` is not weighted, we set all weights to 1 (hence, the output can be any spanning tree).
- `check` – boolean (default: `False`); whether to first perform sanity checks on the input graph `G`. Default: `check=False`. If we toggle `check=True`, the following sanity checks are first performed on `G` prior to running Boruvka's algorithm on that input graph:
  - Is `G` the null graph or graph on one vertex?
  - Is `G` disconnected?
  - Is `G` a tree?

  By default, we turn off the sanity checks for performance reasons. This means that by default the function assumes that its input graph is connected, and has at least one vertex. Otherwise, you should set `check=True` to perform some sanity checks and preprocessing on the input graph.
- `by_weight` – boolean (default: `False`); whether to find MST by using weights of edges provided. Default: `by_weight=True`. If `wfunction` is given, MST is calculated using the weights of edges as per the function. If `wfunction` is `None`, the weight of an edge `e=(u,v,l)` is `l` if graph is weighted, or all edge weights are considered `1` if graph is unweighted. If we toggle `by_weight=False`, all weights are considered as `1` and MST is calculated.

OUTPUT:

The edges of a minimum spanning tree of `G`, if one exists, otherwise returns the empty list.

See also:

- *min_spanning_tree()*

EXAMPLES:

An example from pages 727–728 in [Sah2000]:

```
sage: from sage.graphs.spanning_tree import boruvka
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: G.weighted(True)
```

(continues on next page)

(continued from previous page)

```
sage: E = boruvka(G, check=True); E
[(1, 6, 10), (2, 7, 14), (3, 4, 12), (4, 5, 22), (5, 6, 25), (2, 3, 16)]
sage: boruvka(G, by_weight=True)
[(1, 6, 10), (2, 7, 14), (3, 4, 12), (4, 5, 22), (5, 6, 25), (2, 3, 16)]
sage: sorted(boruvka(G, by_weight=False))
[(1, 2, 28), (1, 6, 10), (2, 3, 16), (2, 7, 14), (3, 4, 12), (4, 5, 22)]
```

An example with custom edge labels:

```
sage: G = Graph([[0,1,1],[1,2,1],[2,0,10]], weighted=True)
sage: weight = lambda e:3-e[0]-e[1]
sage: boruvka(G, wfunction=lambda e:3-e[0]-e[1], by_weight=True)
[(0, 2, 10), (1, 2, 1)]
sage: boruvka(G, wfunction=lambda e:float(1/e[2]), by_weight=True)
[(0, 2, 10), (0, 1, 1)]
```

An example of disconnected graph with `check` disabled:

```
sage: from sage.graphs.spanning_tree import boruvka
sage: G = Graph({1:{2:28}, 3:{4:16}}, weighted=True)
sage: boruvka(G, check=False)
[]
```

sage.graphs.spanning_tree.**filter_kruskal**(*G*, *threshold=10000*, *weight_function=None*, *check=False*)

Minimum spanning tree using Filter Kruskal algorithm.

This function implements the variant of Kruskal's algorithm proposed in [OSS2009]. Instead of directly sorting the whole set of edges, it partitions it in a similar way to quicksort and filter out edges that connect vertices of the same tree to reduce the cost of sorting.

This function assumes that we can only compute minimum spanning trees for undirected graphs. Such graphs can be weighted or unweighted, and they can have multiple edges (since we are computing the minimum spanning tree, only the minimum weight among all (u, v)-edges is considered, for each pair of vertices u, v).

INPUT:

- `G` – an undirected graph
- `weight_function` – function (default: `None`); a function that inputs an edge `e` and outputs its weight. An edge has the form `(u,v,l)`, where `u` and `v` are vertices, `l` is a label (that can be of any kind). The `weight_function` can be used to transform the label into a weight. In particular:
  - if `weight_function` is not `None`, the weight of an edge `e` is `weight_function(e)`;
  - if `weight_function` is `None` (default) and `g` is weighted (that is, `g.weighted()==True`), the weight of an edge `e=(u,v,l)` is `l`, independently on which kind of object `l` is: the ordering of labels relies on Python's operator <;
  - if `weight_function` is `None` and `g` is not weighted, we set all weights to 1 (hence, the output can be any spanning tree).
- **`threshold` – integer (default: 10000); maximum number of edges on** which to run kruskal algorithm. Above that value, edges are partitioned into sets of size at most `threshold`
- `check` – boolean (default: `False`); whether to first perform sanity checks on the input graph `G`. Default: `check=False`. If we toggle `check=True`, the following sanity checks are first performed on `G` prior to running Kruskal's algorithm on that input graph:
  - Is `G` the null graph?

- Is G disconnected?
- Is G a tree?
- Does G have self-loops?
- Does G have multiple edges?

OUTPUT:

The edges of a minimum spanning tree of G, if one exists, otherwise returns the empty list.

See also:

- *sage.graphs.generic_graph.GenericGraph.min_spanning_tree()*
- Wikipedia article Kruskal's_algorithm
- *kruskal()*
- *filter_kruskal_iterator()*

EXAMPLES:

```
sage: from sage.graphs.spanning_tree import filter_kruskal
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: G.weighted(True)
sage: filter_kruskal(G, check=True)
[(1, 6, 10), (3, 4, 12), (2, 7, 14), (2, 3, 16), (4, 5, 22), (5, 6, 25)]

sage: filter_kruskal(Graph(2), check=True)
[]
```

sage.graphs.spanning_tree.**filter_kruskal_iterator**(*G*, *threshold=10000*, *weight_function=None*, *check=False*)

Return an iterator implementation of Filter Kruskal's algorithm.

OUTPUT:

The edges of a minimum spanning tree of G, one by one.

See also:

- *sage.graphs.generic_graph.GenericGraph.min_spanning_tree()*
- Wikipedia article Kruskal's_algorithm
- *kruskal()*
- *filter_kruskal()*

EXAMPLES:

The edges of a minimum spanning tree of G, if one exists, otherwise returns the empty list.

sage: from sage.graphs.spanning_tree import filter_kruskal_iterator sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:{6:25, 7:24}}) sage: G.weighted(True) sage: list(filter_kruskal_iterator(G, threshold=3, check=True)) [(1, 6, 10), (3, 4, 12), (2, 7, 14), (2, 3, 16), (4, 5, 22), (5, 6, 25)]

The weights of the spanning trees returned by *kruskal_iterator()* and *filter_kruskal_iterator()* are the same:

```
sage: from sage.graphs.spanning_tree import kruskal_iterator
sage: G = graphs.RandomBarabasiAlbert(50, 2)
sage: for u, v in G.edge_iterator(labels=False):
....:     G.set_edge_label(u, v, randint(1, 10))
sage: G.weighted(True)
sage: sum(e[2] for e in kruskal_iterator(G)) == sum(e[2] for e in filter_kruskal_
↪iterator(G, threshold=20))
True
```

sage.graphs.spanning_tree.**kruskal**(*G*, *wfunction=None*, *check=False*)

Minimum spanning tree using Kruskal's algorithm.

This function assumes that we can only compute minimum spanning trees for undirected graphs. Such graphs can be weighted or unweighted, and they can have multiple edges (since we are computing the minimum spanning tree, only the minimum weight among all (u, v)-edges is considered, for each pair of vertices u, v).

INPUT:

- `G` – an undirected graph.
- `weight_function` – function (default: `None`); a function that inputs an edge `e` and outputs its weight. An edge has the form `(u,v,l)`, where `u` and `v` are vertices, `l` is a label (that can be of any kind). The `weight_function` can be used to transform the label into a weight. In particular:
  - if `weight_function` is not `None`, the weight of an edge `e` is `weight_function(e)`;
  - if `weight_function` is `None` (default) and `g` is weighted (that is, `g.weighted()==True`), the weight of an edge `e=(u,v,l)` is `l`, independently on which kind of object `l` is: the ordering of labels relies on Python's operator <;
  - if `weight_function` is `None` and `g` is not weighted, we set all weights to 1 (hence, the output can be any spanning tree).
- `check` – boolean (default: `False`); whether to first perform sanity checks on the input graph `G`. Default: `check=False`. If we toggle `check=True`, the following sanity checks are first performed on `G` prior to running Kruskal's algorithm on that input graph:
  - Is `G` the null graph?
  - Is `G` disconnected?
  - Is `G` a tree?
  - Does `G` have self-loops?
  - Does `G` have multiple edges?

  By default, we turn off the sanity checks for performance reasons. This means that by default the function assumes that its input graph is connected, and has at least one vertex. Otherwise, you should set `check=True` to perform some sanity checks and preprocessing on the input graph. If `G` has multiple edges or self-loops, the algorithm still works, but the running-time can be improved if these edges are removed. To further improve the runtime of this function, you should call it directly instead of using it indirectly via *sage.graphs.generic_graph.GenericGraph.min_spanning_tree()*.

OUTPUT:

The edges of a minimum spanning tree of `G`, if one exists, otherwise returns the empty list.

See also:

- *sage.graphs.generic_graph.GenericGraph.min_spanning_tree()*
- *kruskal_iterator()*

- *filter_kruskal()* and *filter_kruskal_iterator()*

EXAMPLES:

An example from pages 727–728 in [Sah2000].

```
sage: from sage.graphs.spanning_tree import kruskal
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: G.weighted(True)
sage: E = kruskal(G, check=True); E
[(1, 6, 10), (3, 4, 12), (2, 7, 14), (2, 3, 16), (4, 5, 22), (5, 6, 25)]
```

Variants of the previous example.

```
sage: H = Graph(G.edges(labels=False))
sage: kruskal(H, check=True)
[(1, 2, None), (1, 6, None), (2, 3, None), (2, 7, None), (3, 4, None), (4, 5,␣
↪None)]
sage: G.allow_loops(True)
sage: G.allow_multiple_edges(True)
sage: G
Looped multi-graph on 7 vertices
sage: for i in range(20):
....:     u = randint(1, 7)
....:     v = randint(1, 7)
....:     w = randint(0, 20)
....:     G.add_edge(u, v, w)
sage: H = copy(G)
sage: H
Looped multi-graph on 7 vertices
sage: def sanitize(G):
....:     G.allow_loops(False)
....:     G.allow_multiple_edges(False, keep_label='min')
sage: sanitize(H)
sage: H
Graph on 7 vertices
sage: sum(e[2] for e in kruskal(G, check=True)) == sum(e[2] for e in kruskal(H,␣
↪check=True))
True
```

An example from pages 599–601 in [GT2001].

```
sage: G = Graph({"SFO":{"BOS":2704, "ORD":1846, "DFW":1464, "LAX":337},
....: "BOS":{"ORD":867, "JFK":187, "MIA":1258},
....: "ORD":{"PVD":849, "JFK":740, "BWI":621, "DFW":802},
....: "DFW":{"JFK":1391, "MIA":1121, "LAX":1235},
....: "LAX":{"MIA":2342},
....: "PVD":{"JFK":144},
....: "JFK":{"MIA":1090, "BWI":184},
....: "BWI":{"MIA":946}})
sage: G.weighted(True)
sage: kruskal(G, check=True)
[('JFK', 'PVD', 144), ('BWI', 'JFK', 184), ('BOS', 'JFK', 187), ('LAX', 'SFO',␣
↪337), ('BWI', 'ORD', 621), ('DFW', 'ORD', 802), ('BWI', 'MIA', 946), ('DFW',
↪'LAX', 1235)]
```

An example from pages 568–569 in [CLRS2001].

```
sage: G = Graph({"a":{"b":4, "h":8}, "b":{"c":8, "h":11},
....: "c":{"d":7, "f":4, "i":2}, "d":{"e":9, "f":14},
....: "e":{"f":10}, "f":{"g":2}, "g":{"h":1, "i":6}, "h":{"i":7}})
sage: G.weighted(True)
sage: T = Graph(kruskal(G, check=True), format='list_of_edges')
sage: sum(T.edge_labels())
37
sage: T.is_tree()
True
```

An example with custom edge labels:

```
sage: G = Graph([[0,1,1],[1,2,1],[2,0,10]], weighted=True)
sage: weight = lambda e:3-e[0]-e[1]
sage: sorted(kruskal(G, check=True))
[(0, 1, 1), (1, 2, 1)]
sage: sorted(kruskal(G, wfunction=weight, check=True))
[(0, 2, 10), (1, 2, 1)]
sage: sorted(kruskal(G, wfunction=weight, check=False))
[(0, 2, 10), (1, 2, 1)]
```

sage.graphs.spanning_tree.**kruskal_iterator**(*G*, *wfunction=None*, *check=False*)

Return an iterator implementation of Kruskal algorithm.

OUTPUT:

The edges of a minimum spanning tree of `G`, one by one.

See also:

kruskal()

EXAMPLES:

```
sage: from sage.graphs.spanning_tree import kruskal_iterator
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: G.weighted(True)
sage: next(kruskal_iterator(G, check=True))
(1, 6, 10)
```

sage.graphs.spanning_tree.**kruskal_iterator_from_edges**(*edges*, *union_find*, *weighted=False*, *weight_function=None*)

Return an iterator implementation of Kruskal algorithm on list of edges.

INPUT:

- `edges` – list of edges
- `union_find` – a `DisjointSet_of_hashables` encoding a forest
- `weighted` – boolean (default: `False`); whether edges are weighted, i.e., the label of an edge is a weight
- `weight_function` – function (default: `None`); a function that inputs an edge `e` and outputs its weight. See *kruskal()* for more details.

OUTPUT:

The edges of a minimum spanning tree of `G`, one by one.

See also:

- *kruskal()*
- *filter_kruskal()*

EXAMPLES:

```
sage: from sage.graphs.spanning_tree import kruskal_iterator_from_edges
sage: G = Graph({1:{2:28, 6:10}, 2:{3:16, 7:14}, 3:{4:12}, 4:{5:22, 7:18}, 5:
↪{6:25, 7:24}})
sage: G.weighted(True)
sage: union_set=DisjointSet(G.order())
sage: next(kruskal_iterator_from_edges(G.edges(sort=False), union_set, weighted=G.
↪weighted()))
(1, 6, 10)
```

sage.graphs.spanning_tree.**random_spanning_tree**(*self*, *output_as_graph=False*)

Return a random spanning tree of the graph.

This uses the Aldous-Broder algorithm ([Bro1989], [Ald1990]) to generate a random spanning tree with the uniform distribution, as follows.

Start from any vertex. Perform a random walk by choosing at every step one neighbor uniformly at random. Every time a new vertex j is met, add the edge (i, j) to the spanning tree, where i is the previous vertex in the random walk.

INPUT:

- `output_as_graph` – boolean (default: `False`); whether to return a list of edges or a graph

See also:

spanning_trees_count() and *spanning_trees()*

EXAMPLES:

```
sage: G = graphs.TietzeGraph()
sage: G.random_spanning_tree(output_as_graph=True)
Graph on 12 vertices
sage: rg = G.random_spanning_tree(); rg # random
[(0, 9),
(9, 11),
(0, 8),
(8, 7),
(7, 6),
(7, 2),
(2, 1),
(1, 5),
(9, 10),
(5, 4),
(2, 3)]
sage: Graph(rg).is_tree()
True
```

A visual example for the grid graph:

```
sage: G = graphs.Grid2dGraph(6, 6)
sage: pos = G.get_pos()
sage: T = G.random_spanning_tree(True)
sage: T.set_pos(pos)
sage: T.show(vertex_labels=False)
```

5.9 PQ-Trees

This module implements PQ-Trees, a data structure use to represent all permutations of the columns of a matrix which satisfy the *consecutive ones property*:

> A binary matrix satisfies the *consecutive ones property* if the 1s are contiguous in each of its rows (or equivalently, if no row contains the regexp pattern 10^+1).
>
> Alternatively, one can say that a sequence of sets $S_1, ..., S_n$ satisfies the *consecutive ones property* if for any x the indices of the sets containing x is an interval of $[1, n]$.

This module is used for the recognition of Interval Graphs (see `is_interval()`).

P-tree and Q-tree

- A P-tree with children $c_1, ..., c_k$ (which can be P-trees, Q-trees, or actual sets of points) indicates that all $k!$ permutations of the children are allowed.

  Example: $\{1, 2\}, \{3, 4\}, \{5, 6\}$ (disjoint sets can be permuted in any way)

- A Q-tree with children $c_1, ..., c_k$ (which can be P-trees, Q-trees, or actual sets of points) indicates that only two permutations of its children are allowed: $c_1, ..., c_k$ or $c_k, ..., c_1$.

  Example: $\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 6\}$ (only two permutations of these sets have the *consecutive ones property*).

Computation of all possible orderings

1. In order to compute all permutations of a sequence of sets $S_1, ..., S_k$ satisfying the *consecutive ones property*, we initialize T as a P-tree whose children are all the $S_1, ..., S_k$, thus representing the set of all $k!$ permutations of them.
2. We select some element x and update the data structure T to restrict the permutations it describes to those that keep the occurrences of x on an interval of $[1, ..., k]$. This will result in a new P-tree whose children are:
   - all $\bar{c}_x$ sets S_i which do *not* contain x.
   - a new P-tree whose children are the c_x sets S_i containing x.

   This describes the set of all $c_x! \times \bar{c}'_x!$ permutations of $S_1, ..., S_k$ that keep the sets containing x on an interval.
3. We take a second element x' and update the data structure T to restrict the permutations it describes to those that keep x' on an interval of $[1, ..., k]$. The sets $S_1, ..., S_k$ belong to 4 categories:
   - The family S_{00} of sets which do not contain any of x, x'.
   - The family S_{01} of sets which contain x' but do not contain x.
   - The family S_{10} of sets which contain x but do not contain x'.
   - The family S_{11} of sets which contain x' and x'.

   With these notations, the permutations of $S_1, ..., S_k$ which keep the occurrences of x and x' on an interval are of two forms:
   - <some sets S_{00}>, <sets from S_{10}>, <sets from S_{11}>, <sets from S_{01}>, <other sets from S_{00}>
   - <some sets S_{00}>, <sets from S_{01}>, <sets from S_{11}>, <sets from S_{10}>, <other sets from S_{00}>

   These permutations can be modeled with the following PQ-tree:
   - A P-tree whose children are:
     - All sets from S_{00}
     - A Q-tree whose children are:

- A P-tree with whose children are the sets from S_{10}
- A P-tree with whose children are the sets from S_{11}
- A P-tree with whose children are the sets from S_{01}

4. One at a time, we update the data structure with each element until they are all exhausted, or until we reach a proof that no permutation satisfying the *consecutive ones property* exists.

   Using these two types of tree, and exploring the different cases of intersection, it is possible to represent all the possible permutations of our sets satisfying our constraints, or to prove that no such ordering exists. This is the whole purpose of this module, and is explained with more details in many places, for example in the following document from Hajiaghayi [Haj2000].

Authors:

Nathann Cohen (initial implementation)

5.9.1 Methods and functions

class sage.graphs.pq_trees.**P**(*seq*)
Bases: *sage.graphs.pq_trees.PQ*

A P-Tree is a PQ-Tree whose children can be permuted in any way.

For more information, see the documentation of *sage.graphs.pq_trees*.

cardinality()
Return the number of orderings allowed by the structure.

See also:

orderings() – iterate over all admissible orderings

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = P([[0,3], [1,2], [2,3], [2,4], [4,0],[2,8], [2,9]])
sage: p.cardinality()
5040
sage: p.set_contiguous(3)
(1, True)
sage: p.cardinality()
1440
```

orderings()
Iterate over all orderings of the sets allowed by the structure.

See also:

cardinality() – return the number of orderings

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = P([[2,4], [1,2], [0,8], [0,5]])
sage: for o in p.orderings():
....:     print(o)
({2, 4}, {1, 2}, {0, 8}, {0, 5})
({2, 4}, {1, 2}, {0, 5}, {0, 8})
({2, 4}, {0, 8}, {1, 2}, {0, 5})
```

(continues on next page)

(continued from previous page)

```
({2, 4}, {0, 8}, {0, 5}, {1, 2})
...
```

set_contiguous(*v*)

Updates `self` so that the sets containing `v` are contiguous for any admissible permutation of its subtrees.

INPUT:

- `v` – an element of the ground set

OUTPUT:

According to the cases :

- `(EMPTY, ALIGNED)` if no set of the tree contains an occurrence of `v`
- `(FULL, ALIGNED)` if all the sets of the tree contain `v`
- `(PARTIAL, ALIGNED)` if some (but not all) of the sets contain `v`, all of which are aligned to the right of the ordering at the end when the function ends
- `(PARTIAL, UNALIGNED)` if some (but not all) of the sets contain `v`, though it is impossible to align them all to the right

In any case, the sets containing `v` are contiguous when this function ends. If there is no possibility of doing so, the function raises a `ValueError` exception.

EXAMPLES:

Ensuring the sets containing `0` are continuous:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = P([[0,3], [1,2], [2,3], [2,4], [4,0],[2,8], [2,9]])
sage: p.set_contiguous(0)
(1, True)
sage: print(p)
('P', [{1, 2}, {2, 3}, {2, 4}, {8, 2}, {9, 2}, ('P', [{0, 3}, {0, 4}])])
```

Impossible situation:

```
sage: p = P([[0,1], [1,2], [2,3], [3,0]])
sage: p.set_contiguous(0)
(1, True)
sage: p.set_contiguous(1)
(1, True)
sage: p.set_contiguous(2)
(1, True)
sage: p.set_contiguous(3)
Traceback (most recent call last):
...
ValueError: Impossible
```

class sage.graphs.pq_trees.**PQ**(*seq*)

Bases: `object`

PQ-Trees

This class should not be instantiated by itself: it is extended by *P* and *Q*. See the documentation of *sage.graphs.pq_trees* for more information.

AUTHOR : Nathann Cohen

flatten()
Returns a flattened copy of `self`

If self has only one child, we may as well consider its child's children, as `self` encodes no information. This method recursively "flattens" trees having only on PQ-tree child, and returns it.

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = Q([P([[2,4], [2,8], [2,9]])])
sage: p.flatten()
('P', [{2, 4}, {8, 2}, {9, 2}])
```

number_of_children()
Returns the number of children of `self`

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = Q([[1,2], [2,3], P([[2,4], [2,8], [2,9]])])
sage: p.number_of_children()
3
```

ordering()
Returns the current ordering given by listing the leaves from left to right.

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = Q([[1,2], [2,3], P([[2,4], [2,8], [2,9]])])
sage: p.ordering()
[{1, 2}, {2, 3}, {2, 4}, {8, 2}, {9, 2}]
```

reverse()
Recursively reverses `self` and its children

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = Q([[1,2], [2,3], P([[2,4], [2,8], [2,9]])])
sage: p.ordering()
[{1, 2}, {2, 3}, {2, 4}, {8, 2}, {9, 2}]
sage: p.reverse()
sage: p.ordering()
[{9, 2}, {8, 2}, {2, 4}, {2, 3}, {1, 2}]
```

simplify(*v*, *left=False*, *right=False*)
Returns a simplified copy of self according to the element `v`

If `self` is a partial P-tree for `v`, we would like to restrict the permutations of its children to permutations keeping the children containing `v` contiguous. This function also "locks" all the elements not containing `v` inside a P-tree, which is useful when one want to keep the elements containing `v` on one side (which is the case when this method is called).

INPUT:

- `left, right` (boolean) – whether `v` is aligned to the right or to the left
- `v`– an element of the ground set

OUTPUT:

If `self` is a Q-Tree, the sequence of its children is returned. If `self` is a P-tree, 2 P-tree are returned, namely the two P-tree defined above and restricting the permutations, in the order implied by `left, right` (if `right =True`, the second P-tree will be the one gathering the elements containing `v`, if `left=True`, the opposite).

Note: This method is assumes that `self` is partial for `v`, and aligned to the side indicated by `left, right`.

EXAMPLES:

A P-Tree

```
sage: from sage.graphs.pq_trees import P, Q
sage: p = P([[2,4], [1,2], [0,8], [0,5]])
sage: p.simplify(0, right = True)
[('P', [{2, 4}, {1, 2}]), ('P', [{0, 8}, {0, 5}])]
```

A Q-Tree

```
sage: q = Q([[2,4], [1,2], [0,8], [0,5]])
sage: q.simplify(0, right = True)
[{2, 4}, {1, 2}, {0, 8}, {0, 5}]
```

class sage.graphs.pq_trees.**Q**(*seq*)

Bases: *sage.graphs.pq_trees.PQ*

A Q-Tree is a PQ-Tree whose children are ordered up to reversal

For more information, see the documentation of *sage.graphs.pq_trees*.

cardinality()

Return the number of orderings allowed by the structure.

See also:

orderings() – iterate over all admissible orderings

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: q = Q([[0,3], [1,2], [2,3], [2,4], [4,0],[2,8], [2,9]])
sage: q.cardinality()
2
```

orderings()

Iterates over all orderings of the sets allowed by the structure

See also:

cardinality() – return the number of orderings

EXAMPLES:

```
sage: from sage.graphs.pq_trees import P, Q
sage: q = Q([[2,4], [1,2], [0,8], [0,5]])
sage: for o in q.orderings():
....:     print(o)
({2, 4}, {1, 2}, {0, 8}, {0, 5})
({0, 5}, {0, 8}, {1, 2}, {2, 4})
```

set_contiguous(*v*)

Updates `self` so that the sets containing `v` are contiguous for any admissible permutation of its subtrees.

INPUT:

- `v` – an element of the ground set

OUTPUT:

According to the cases :

- `(EMPTY, ALIGNED)` if no set of the tree contains an occurrence of `v`
- `(FULL, ALIGNED)` if all the sets of the tree contain `v`
- `(PARTIAL, ALIGNED)` if some (but not all) of the sets contain `v`, all of which are aligned to the right of the ordering at the end when the function ends
- `(PARTIAL, UNALIGNED)` if some (but not all) of the sets contain `v`, though it is impossible to align them all to the right

In any case, the sets containing `v` are contiguous when this function ends. If there is no possibility of doing so, the function raises a `ValueError` exception.

EXAMPLES:

Ensuring the sets containing `0` are continuous:

```
sage: from sage.graphs.pq_trees import P, Q
sage: q = Q([[2,3], Q([[3,0],[3,1]]), Q([[4,0],[4,5]])])
sage: q.set_contiguous(0)
(1, False)
sage: print(q)
('Q', [{2, 3}, {1, 3}, {0, 3}, {0, 4}, {4, 5}])
```

Impossible situation:

```
sage: p = Q([[0,1], [1,2], [2,0]])
sage: p.set_contiguous(0)
Traceback (most recent call last):
...
ValueError: Impossible
```

sage.graphs.pq_trees.**flatten**(*x*)

sage.graphs.pq_trees.**new_P**(*liste*)

sage.graphs.pq_trees.**new_Q**(*liste*)

sage.graphs.pq_trees.**reorder_sets**(*sets*)

Reorders a collection of sets such that each element appears on an interval.

Given a collection of sets $C = S_1, ..., S_k$ on a ground set X, this function attempts to reorder them in such a way that $\forall x \in X$ and $i < j$ with $x \in S_i, S_j$, then $x \in S_l$ for every $i < l < j$ if it exists.

INPUT:

- `sets` - a list of instances of `list, Set` or `set`

ALGORITHM:

PQ-Trees

EXAMPLES:

There is only one way (up to reversal) to represent contiguously the sequence ofsets $\{i-1, i, i+1\}$:

```
sage: from sage.graphs.pq_trees import reorder_sets
sage: seq = [Set([i-1,i,i+1]) for i in range(1,15)]
```

We apply a random permutation:

```
sage: p = Permutations(len(seq)).random_element()
sage: seq = [ seq[p(i+1)-1] for i in range(len(seq)) ]
sage: ordered = reorder_sets(seq)
sage: if not 0 in ordered[0]:
....:     ordered = ordered.reverse()
sage: print(ordered)
[{0, 1, 2}, {1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}, {5, 6, 7}, {8, 6, 7}, {8,␣
↪9, 7}, {8, 9, 10}, {9, 10, 11}, {10, 11, 12}, {11, 12, 13}, {12, 13, 14}, {13,␣
↪14, 15}]
```

sage.graphs.pq_trees.**set_contiguous**(*tree*, *x*)

5.10 Generation of trees

This is an implementation of the algorithm for generating trees with n vertices (up to isomorphism) in constant time per tree described in [WROM1986].

AUTHORS:

- Ryan Dingman (2009-04-16): initial version

class sage.graphs.trees.**TreeIterator**

Bases: object

This class iterates over all trees with n vertices (up to isomorphism).

EXAMPLES:

```
sage: from sage.graphs.trees import TreeIterator
sage: def check_trees(n):
....:     trees = []
....:     for t in TreeIterator(n):
....:         if not t.is_tree():
....:             return False
....:         if t.num_verts() != n:
....:             return False
....:         if t.num_edges() != n - 1:
....:             return False
....:         for tree in trees:
....:             if tree.is_isomorphic(t):
....:                 return False
....:         trees.append(t)
....:     return True
sage: check_trees(10)
True
```

```
sage: from sage.graphs.trees import TreeIterator
sage: count = 0
sage: for t in TreeIterator(15):
....:     count += 1
sage: count
7741
```

5.11 Matching Polynomial

This module contains the following methods:

matching_polynomial()	Computes the matching polynomial of a given graph
complete_poly()	Compute the matching polynomial of the complete graph on n vertices.

AUTHORS:

- Robert Miller, Tom Boothby - original implementation

REFERENCE:

[God1993]

5.11.1 Methods

sage.graphs.matchpoly.**complete_poly**(n)

Compute the matching polynomial of the complete graph on n vertices.

INPUT:

- `n` – order of the complete graph

Todo: This code could probably be made more efficient by using FLINT polynomials and being written in Cython, using an array of fmpz_poly_t pointers or something... Right now just about the whole complement optimization is written in Python, and could be easily sped up.

EXAMPLES:

```
sage: from sage.graphs.matchpoly import complete_poly
sage: f = complete_poly(10)
sage: f
x^10 - 45*x^8 + 630*x^6 - 3150*x^4 + 4725*x^2 - 945
sage: f = complete_poly(20)
sage: f[8]
1309458150
sage: f = complete_poly(1000)
sage: len(str(f))
406824
```

sage.graphs.matchpoly.**matching_polynomial**(G, *complement=True*, *name=None*)

Computes the matching polynomial of the graph G.

If $p(G, k)$ denotes the number of k-matchings (matchings with k edges) in G, then the matching polynomial is defined as [God1993]:

$$\mu(x) = \sum_{k \geq 0} (-1)^k p(G, k) x^{n-2k}$$

INPUT:

- `complement` - (default: `True`) whether to use Godsil's duality theorem to compute the matching polynomial from that of the graphs complement (see ALGORITHM).
- `name` - optional string for the variable name in the polynomial

Note: The `complement` option uses matching polynomials of complete graphs, which are cached. So if you are crazy enough to try computing the matching polynomial on a graph with millions of vertices, you might not want to use this option, since it will end up caching millions of polynomials of degree in the millions.

ALGORITHM:

The algorithm used is a recursive one, based on the following observation [God1993]:

- If e is an edge of G, G' is the result of deleting the edge e, and G'' is the result of deleting each vertex in e, then the matching polynomial of G is equal to that of G' minus that of G''.

  (the algorithm actually computes the *signless* matching polynomial, for which the recursion is the same when one replaces the substraction by an addition. It is then converted into the matching polynomial and returned)

Depending on the value of `complement`, Godsil's duality theorem [God1993] can also be used to compute $\mu(x)$:

$$\mu(\overline{G}, x) = \sum_{k \geq 0} p(G, k)\mu(K_{n-2k}, x)$$

Where $\overline{G}$ is the complement of G, and K_n the complete graph on n vertices.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.matching_polynomial()
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(complement=False)
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(name='tom')
tom^10 - 15*tom^8 + 75*tom^6 - 145*tom^4 + 90*tom^2 - 6
sage: g = Graph()
sage: L = [graphs.RandomGNP(8, .3) for i in range(1, 6)]
sage: prod([h.matching_polynomial() for h in L]) == sum(L, g).matching_
↪polynomial()  # long time (up to 10s on sage.math, 2011)
True
```

```
sage: for i in range(1, 12):  # long time (10s on sage.math, 2011)
....:     for t in graphs.trees(i):
....:         if t.matching_polynomial() != t.characteristic_polynomial():
....:             raise RuntimeError('bug for a tree A of size {0}'.format(i))
....:         c = t.complement()
....:         if c.matching_polynomial(complement=False) != c.matching_
↪polynomial():
....:             raise RuntimeError('bug for a tree B of size {0}'.format(i))
```

```
sage: from sage.graphs.matchpoly import matching_polynomial
sage: matching_polynomial(graphs.CompleteGraph(0))
1
sage: matching_polynomial(graphs.CompleteGraph(1))
x
sage: matching_polynomial(graphs.CompleteGraph(2))
x^2 - 1
sage: matching_polynomial(graphs.CompleteGraph(3))
x^3 - 3*x
sage: matching_polynomial(graphs.CompleteGraph(4))
```

(continues on next page)

(continued from previous page)

```
x^4 - 6*x^2 + 3
sage: matching_polynomial(graphs.CompleteGraph(5))
x^5 - 10*x^3 + 15*x
sage: matching_polynomial(graphs.CompleteGraph(6))
x^6 - 15*x^4 + 45*x^2 - 15
sage: matching_polynomial(graphs.CompleteGraph(7))
x^7 - 21*x^5 + 105*x^3 - 105*x
sage: matching_polynomial(graphs.CompleteGraph(8))
x^8 - 28*x^6 + 210*x^4 - 420*x^2 + 105
sage: matching_polynomial(graphs.CompleteGraph(9))
x^9 - 36*x^7 + 378*x^5 - 1260*x^3 + 945*x
sage: matching_polynomial(graphs.CompleteGraph(10))
x^10 - 45*x^8 + 630*x^6 - 3150*x^4 + 4725*x^2 - 945
sage: matching_polynomial(graphs.CompleteGraph(11))
x^11 - 55*x^9 + 990*x^7 - 6930*x^5 + 17325*x^3 - 10395*x
sage: matching_polynomial(graphs.CompleteGraph(12))
x^12 - 66*x^10 + 1485*x^8 - 13860*x^6 + 51975*x^4 - 62370*x^2 + 10395
sage: matching_polynomial(graphs.CompleteGraph(13))
x^13 - 78*x^11 + 2145*x^9 - 25740*x^7 + 135135*x^5 - 270270*x^3 + 135135*x
```

```
sage: G = Graph({0:[1,2], 1:[2]})
sage: matching_polynomial(G)
x^3 - 3*x
sage: G = Graph({0:[1,2]})
sage: matching_polynomial(G)
x^3 - 2*x
sage: G = Graph({0:[1], 2:[]})
sage: matching_polynomial(G)
x^3 - x
sage: G = Graph({0:[], 1:[], 2:[]})
sage: matching_polynomial(G)
x^3
```

```
sage: matching_polynomial(graphs.CompleteGraph(0), complement=False)
1
sage: matching_polynomial(graphs.CompleteGraph(1), complement=False)
x
sage: matching_polynomial(graphs.CompleteGraph(2), complement=False)
x^2 - 1
sage: matching_polynomial(graphs.CompleteGraph(3), complement=False)
x^3 - 3*x
sage: matching_polynomial(graphs.CompleteGraph(4), complement=False)
x^4 - 6*x^2 + 3
sage: matching_polynomial(graphs.CompleteGraph(5), complement=False)
x^5 - 10*x^3 + 15*x
sage: matching_polynomial(graphs.CompleteGraph(6), complement=False)
x^6 - 15*x^4 + 45*x^2 - 15
sage: matching_polynomial(graphs.CompleteGraph(7), complement=False)
x^7 - 21*x^5 + 105*x^3 - 105*x
sage: matching_polynomial(graphs.CompleteGraph(8), complement=False)
x^8 - 28*x^6 + 210*x^4 - 420*x^2 + 105
sage: matching_polynomial(graphs.CompleteGraph(9), complement=False)
x^9 - 36*x^7 + 378*x^5 - 1260*x^3 + 945*x
sage: matching_polynomial(graphs.CompleteGraph(10), complement=False)
x^10 - 45*x^8 + 630*x^6 - 3150*x^4 + 4725*x^2 - 945
sage: matching_polynomial(graphs.CompleteGraph(11), complement=False)
```

(continues on next page)

(continued from previous page)

```
x^11 - 55*x^9 + 990*x^7 - 6930*x^5 + 17325*x^3 - 10395*x
sage: matching_polynomial(graphs.CompleteGraph(12), complement=False)
x^12 - 66*x^10 + 1485*x^8 - 13860*x^6 + 51975*x^4 - 62370*x^2 + 10395
sage: matching_polynomial(graphs.CompleteGraph(13), complement=False)
x^13 - 78*x^11 + 2145*x^9 - 25740*x^7 + 135135*x^5 - 270270*x^3 + 135135*x
```

5.12 Genus

This file contains a moderately-optimized implementation to compute the genus of simple connected graph. It runs about a thousand times faster than the previous version in Sage, not including asymptotic improvements.

The algorithm works by enumerating combinatorial embeddings of a graph, and computing the genus of these via the Euler characteristic. We view a combinatorial embedding of a graph as a pair of permutations v, e which act on a set B of $2|E(G)|$ "darts". The permutation e is an involution, and its orbits correspond to edges in the graph. Similarly, The orbits of v correspond to the vertices of the graph, and those of $f = ve$ correspond to faces of the embedded graph.

The requirement that the group $< v, e >$ acts transitively on B is equivalent to the graph being connected. We can compute the genus of a graph by

$$2 - 2g = V - E + F$$

where E, V, and F denote the number of orbits of e, v, and f respectively.

We make several optimizations to the naive algorithm, which are described throughout the file.

class sage.graphs.genus.**simple_connected_genus_backtracker**

Bases: object

A class which computes the genus of a DenseGraph through an extremely slow but relatively optimized algorithm. This is "only" exponential for graphs of bounded degree, and feels pretty snappy for 3-regular graphs. The generic runtime is

$$|V(G)| \prod_{v \in V(G)} (deg(v) - 1)!$$

which is $2^{|V(G)|}$ for 3-regular graphs, and can achieve $n(n-1)!^n$ for the complete graph on n vertices. We can handily compute the genus of K_6 in milliseconds on modern hardware, but K_7 may take a few days. Don't bother with K_8, or any graph with more than one vertex of degree 10 or worse, unless you can find an a priori lower bound on the genus and expect the graph to have that genus.

Warning:

THIS MAY SEGFAULT OR HANG ON:

- DISCONNECTED GRAPHS
- DIRECTED GRAPHS
- LOOPED GRAPHS
- MULTIGRAPHS

EXAMPLES:

```
sage: import sage.graphs.genus
sage: G = graphs.CompleteGraph(6)
sage: G = Graph(G, sparse=False)
```

(continues on next page)

(continued from previous page)

```
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: bt.genus() #long time
1
sage: bt.genus(cutoff=1)
1
sage: G = graphs.PetersenGraph()
sage: G = Graph(G, sparse=False)
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: bt.genus()
1
sage: G = graphs.FlowerSnark()
sage: G = Graph(G, sparse=False)
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: bt.genus()
2
```

genus (*style=1*, *cutoff=0*, *record_embedding=False*)

Compute the minimal or maximal genus of self's graph.

Note, this is a remarkably naive algorithm for a very difficult problem. Most interesting cases will take millenia to finish, with the exception of graphs with max degree 3.

INPUT:

- `style` – integer (default: `1`); find minimum genus if 1, maximum genus if 2
- `cutoff` – integer (default: `0`); stop searching if search style is 1 and `genus` $\leq$ `cutoff`, or if style is 2 and `genus` $\geq$ `cutoff`. This is useful where the genus of the graph has a known bound.
- `record_embedding` – boolean (default: `False`); whether or not to remember the best embedding seen. This embedding can be retrieved with `self.get_embedding()`.

OUTPUT:

the minimal or maximal genus for self's graph.

EXAMPLES:

```
sage: import sage.graphs.genus
sage: G = Graph(graphs.CompleteGraph(5), sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: gb.genus(cutoff=2, record_embedding=True)
2
sage: E = gb.get_embedding()
sage: gb.genus(record_embedding=False)
1
sage: gb.get_embedding() == E
True
sage: gb.genus(style=2, cutoff=5)
3
sage: G = Graph(sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: gb.genus()
0
```

get_embedding()

Return an embedding for the graph.

If `min_genus_backtrack` has been called with `record_embedding = True`, then this will return the first minimal embedding that we found. Otherwise, this returns the first embedding considered.

EXAMPLES:

```
sage: import sage.graphs.genus
sage: G = Graph(graphs.CompleteGraph(5), sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: gb.genus(record_embedding=True)
1
sage: gb.get_embedding()
{0: [1, 2, 3, 4], 1: [0, 2, 3, 4], 2: [0, 1, 4, 3], 3: [0, 2, 1, 4], 4: [0, 3,
↪ 1, 2]}
sage: G = Graph(sparse=False)
sage: G.add_edge(0,1)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: gb.get_embedding()
{0: [1], 1: [0]}
sage: G = Graph(sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend.c_
↪graph()[0])
sage: gb.get_embedding()
{}
```

sage.graphs.genus.**simple_connected_graph_genus**(*G*, *set_embedding=False*, *check=True*, *minimal=True*)

Compute the genus of a simple connected graph.

Warning:

THIS MAY SEGFAULT OR HANG ON:

- DISCONNECTED GRAPHS
- DIRECTED GRAPHS
- LOOPED GRAPHS
- MULTIGRAPHS

DO NOT CALL WITH `check = False` UNLESS YOU ARE CERTAIN.

EXAMPLES:

```
sage: import sage.graphs.genus
sage: from sage.graphs.genus import simple_connected_graph_genus as genus
sage: [genus(g) for g in graphs(6) if g.is_connected()].count(1)
13
sage: G = graphs.FlowerSnark()
sage: genus(G)  # see [1]
2
sage: G = graphs.BubbleSortGraph(4)
sage: genus(G)
0
```

(continues on next page)

(continued from previous page)

```
sage: G = graphs.OddGraph(3)
sage: genus(G)
1
```

REFERENCES:

[1] http://www.springerlink.com/content/0776127h0r7548v7/

5.13 Lovász theta-function of graphs

AUTHORS:

- Dima Pasechnik (2015-06-30): Initial version

REFERENCE:

[Lov1979]

5.13.1 Functions

sage.graphs.lovasz_theta.**lovasz_theta**(*graph*)

Return the value of Lovász theta-function of graph

For a graph G this function is denoted by $\theta(G)$, and it can be computed in polynomial time. Mathematically, its most important property is the following:

$$\alpha(G) \leq \theta(G) \leq \chi(\overline{G})$$

with $\alpha(G)$ and $\chi(\overline{G})$ being, respectively, the maximum size of an *independent set* set of G and the *chromatic number* of the *complement* $\overline{G}$ of G.

For more information, see the Wikipedia article Lovász_number.

Note:

- Implemented for undirected graphs only. Use `to_undirected` to convert a digraph to an undirected graph.
- This function requires the optional package `csdp`, which you can install with `sage -i csdp`.

EXAMPLES:

```
sage: C = graphs.PetersenGraph()
sage: C.lovasz_theta()                          # optional csdp
4.0
sage: graphs.CycleGraph(5).lovasz_theta()       # optional csdp
2.236068
```

5.14 Schnyder's Algorithm for straight-line planar embeddings

A module for computing the (x,y) coordinates for a straight-line planar embedding of any connected planar graph with at least three vertices. Uses Walter Schnyder's Algorithm from [Sch1990].

AUTHORS:

- Jonathan Bober, Emily Kirkman (2008-02-09) – initial version

class sage.graphs.schnyder.**TreeNode**(*parent=None*, *children=None*, *label=None*)
Bases: object

A class to represent each node in the trees used by _realizer and _compute_coordinates when finding a planar geometric embedding in the grid.

Each tree node is doubly linked to its parent and children.

INPUT:

- parent – the parent TreeNode of self
- children – a list of TreeNode children of self
- label – the associated realizer vertex label

EXAMPLES:

```
sage: from sage.graphs.schnyder import TreeNode
sage: tn = TreeNode(label=5)
sage: tn2 = TreeNode(label=2,parent=tn)
sage: tn3 = TreeNode(label=3)
sage: tn.append_child(tn3)
sage: tn.compute_number_of_descendants()
2
sage: tn.number_of_descendants
2
sage: tn3.number_of_descendants
1
sage: tn.compute_depth_of_self_and_children()
sage: tn3.depth
2
```

append_child(*child*)
Add a child to list of children.

EXAMPLES:

```
sage: from sage.graphs.schnyder import TreeNode
sage: tn = TreeNode(label=5)
sage: tn2 = TreeNode(label=2,parent=tn)
sage: tn3 = TreeNode(label=3)
sage: tn.append_child(tn3)
sage: tn.compute_number_of_descendants()
2
sage: tn.number_of_descendants
2
sage: tn3.number_of_descendants
1
sage: tn.compute_depth_of_self_and_children()
sage: tn3.depth
2
```

compute_depth_of_self_and_children()
Computes the depth of self and all descendants.

For each TreeNode, sets result as attribute self.depth

EXAMPLES:

```
sage: from sage.graphs.schnyder import TreeNode
sage: tn = TreeNode(label=5)
sage: tn2 = TreeNode(label=2,parent=tn)
sage: tn3 = TreeNode(label=3)
sage: tn.append_child(tn3)
sage: tn.compute_number_of_descendants()
2
sage: tn.number_of_descendants
2
sage: tn3.number_of_descendants
1
sage: tn.compute_depth_of_self_and_children()
sage: tn3.depth
2
```

compute_number_of_descendants()

Computes the number of descendants of self and all descendants.

For each TreeNode, sets result as attribute self.number_of_descendants

EXAMPLES:

```
sage: from sage.graphs.schnyder import TreeNode
sage: tn = TreeNode(label=5)
sage: tn2 = TreeNode(label=2,parent=tn)
sage: tn3 = TreeNode(label=3)
sage: tn.append_child(tn3)
sage: tn.compute_number_of_descendants()
2
sage: tn.number_of_descendants
2
sage: tn3.number_of_descendants
1
sage: tn.compute_depth_of_self_and_children()
sage: tn3.depth
2
```

sage.graphs.schnyder.**minimal_schnyder_wood**(*graph*, *root_edge=None*, *minimal=True*, *check=True*)

Return the minimal Schnyder wood of a planar rooted triangulation.

INPUT:

- graph – a planar triangulation, given by a graph with an embedding.
- root_edge – a pair of vertices (default is from `-1` to `-2`) The third boundary vertex is then determined using the orientation and will be labelled `-3`.
- minimal – boolean (default `True`), whether to return a minimal or a maximal Schnyder wood.
- check – boolean (default `True`), whether to check if the input is a planar triangulation

OUTPUT:

A planar graph, with edges oriented and colored. The three outer edges of the initial graph are removed. For the three outer vertices the list of the neighbors stored in the combinatorial embedding is in the order of the incident edges between the two incident (and removed) outer edges, and not a cyclic shift of it.

The algorithm is taken from [Bre2000] (section 4.2).

EXAMPLES:

```
sage: from sage.graphs.schnyder import minimal_schnyder_wood
sage: g = Graph([(0,-1),(0,-2),(0,-3),(-1,-2),(-2,-3),
....:  (-3,-1)], format='list_of_edges')
sage: g.set_embedding({-1:[-2,0,-3],-2:[-3,0,-1],
....:  -3:[-1,0,-2],0:[-1,-2,-3]})
sage: newg = minimal_schnyder_wood(g)
sage: newg.edges()
[(0, -3, 'red'), (0, -2, 'blue'), (0, -1, 'green')]
sage: newg.plot(color_by_label={'red':'red','blue':'blue',
....:  'green':'green',None:'black'})
Graphics object consisting of 8 graphics primitives
```

A larger example:

```
sage: g = Graph([(0,-1),(0,2),(0,1),(0,-3),(-1,-3),(-1,2),
....: (-1,-2),(1,2),(1,-3),(2,-2),(1,-2),(-2,-3)], format='list_of_edges')
sage: g.set_embedding({-1:[-2,2,0,-3],-2:[-3,1,2,-1],
....: -3:[-1,0,1,-2],0:[-1,2,1,-3],1:[-2,-3,0,2],2:[-1,-2,1,0]})
sage: newg = minimal_schnyder_wood(g)
sage: sorted(newg.edges(), key=lambda e:(str(e[0]),str(e[1])))
[(0, -1, 'green'),
 (0, -3, 'red'),
 (0, 2, 'blue'),
 (1, -2, 'blue'),
 (1, -3, 'red'),
 (1, 0, 'green'),
 (2, -1, 'green'),
 (2, -2, 'blue'),
 (2, 1, 'red')]
sage: newg2 = minimal_schnyder_wood(g, minimal=False)
sage: sorted(newg2.edges(), key=lambda e:(str(e[0]),str(e[1])))
[(0, -1, 'green'),
 (0, -3, 'red'),
 (0, 1, 'blue'),
 (1, -2, 'blue'),
 (1, -3, 'red'),
 (1, 2, 'green'),
 (2, -1, 'green'),
 (2, -2, 'blue'),
 (2, 0, 'red')]
```

5.15 Wrapper for Boyer's (C) planarity algorithm

sage.graphs.planarity.**is_planar**(*g*, *kuratowski=False*, *set_pos=False*, *set_embedding=False*, *circular=False*)

Check whether `g` is planar using Boyer's planarity algorithm.

If `kuratowski` is `False`, returns `True` if `g` is planar, `False` otherwise. If `kuratowski` is `True`, returns a tuple, first entry is a boolean (whether or not the graph is planar) and second entry is a Kuratowski subgraph, i.e. an edge subdivision of K_5 or $K_{3,3}$ (if not planar) or `None` (if planar). Also, will set an `_embedding` attribute for the graph `g` if `set_embedding` is set to `True`.

INPUT:

- `kuratowski` – boolean (default: `False`); when set to `True`, return a tuple of a boolean and either `None` or a Kuratowski subgraph (i.e. an edge subdivision of K_5 or $K_{3,3}$). When set to `False`, returns

`True` if `g` is planar, `False` otherwise.

- `set_pos` – boolean (default: `False`); whether to use Schnyder's algorithm to determine and set positions
- `set_embedding` – boolean (default: `False`); whether to record the combinatorial embedding returned (see *get_embedding()*)
- `circular` – boolean (default: `False`); whether to test for circular planarity

EXAMPLES:

```
sage: G = graphs.DodecahedralGraph()
sage: from sage.graphs.planarity import is_planar
sage: is_planar(G)
True
sage: Graph('@').is_planar()
True
```

5.16 Graph traversals.

This module implements the following graph traversals

lex_BFS()	Perform a lexicographic breadth first search (LexBFS) on the graph.
lex_DFS()	Perform a lexicographic depth first search (LexDFS) on the graph.
lex_UP()	Perform a lexicographic UP search (LexUP) on the graph.
lex_DOWN()	Perform a lexicographic DOWN search (LexDOWN) on the graph.
lex_M()	Return an ordering of the vertices according the LexM graph traversal.
lex_M_slow()	Return an ordering of the vertices according the LexM graph traversal.
lex_M_fast()	Return an ordering of the vertices according the LexM graph traversal.
maximum_cardinality_search()	Return an ordering of the vertices according a maximum cardinality search.
maximum_cardinality_search_M()	Return the ordering and the edges of the triangulation produced by MCS-M.

5.16.1 Methods

`sage.graphs.traversals.`**`is_valid_lex_M_order`**(*G*, *alpha*, *F*)

Check whether the ordering alpha and the triangulation F are valid for G.

Given the graph $G = (V, E)$ with vertex set V and edge set E, and the set F of edges of a triangulation of G, let $H = (V, E \cup F)$. By induction one can see that for every $i \in \{1, ..., n-1\}$ the neighbors of $\alpha(i)$ in $H[\{\alpha(i), ..., \alpha(n)\}]$ induce a clique. The ordering α is a perfect elimination ordering of H, so H is chordal. See [RTL76] for more details.

INPUT:

- `G` – a Graph
- `alpha` – list; an ordering of the vertices of G
- `F` – an iterable of edges given either as `(u, v)` or `(u, v, label)`, the edges of the triangulation of G

`sage.graphs.traversals.`**`lex_BFS`**(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic breadth first search (LexBFS) on the graph.

INPUT:

- `G` – a sage graph

- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated.

Time complexity is $O(n+m)$ where n is the number of vertices and m is the number of edges.

See [CK2008] for more details on the algorithm.

See also:

- Wikipedia article Lexicographic_breadth-first_search
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex BFS is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_BFS()) == g.order()
True
```

Lex BFS ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: g.lex_BFS()
[1, 2, 3, 5, 4, 6]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_BFS(initial_vertex=2)
[2, 3, 1]
```

For a Chordal Graph, a reversed Lex BFS is a Perfect Elimination Order:

```
sage: g = graphs.PathGraph(3).lexicographic_product(graphs.CompleteGraph(2))
sage: g.lex_BFS(reverse=True)  # py2
[(2, 0), (2, 1), (1, 1), (1, 0), (0, 0), (0, 1)]
sage: g.lex_BFS(reverse=True)  # py3
[(2, 1), (2, 0), (1, 1), (1, 0), (0, 1), (0, 0)]
```

And the vertices at the end of the tree of discovery are, for chordal graphs, simplicial vertices (their neighborhood is a complete graph):

```
sage: g = graphs.ClawGraph().lexicographic_product(graphs.CompleteGraph(2))
sage: v = g.lex_BFS()[-1]
sage: peo, tree = g.lex_BFS(initial_vertex = v,  tree=True)
```

(continues on next page)

(continued from previous page)

```
sage: leaves = [v for v in tree if tree.in_degree(v) ==0]
sage: all(g.subgraph(g.neighbors(v)).is_clique() for v in leaves)
True
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

sage.graphs.traversals.**lex_DFS**(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic depth first search (LexDFS) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. Lex DFS differs from Lex BFS only in the way codes are updated after each iteration.

Time complexity is $O(n + m)$ where n is the number of vertices and m is the number of edges.

See [CK2008] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex DFS is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_DFS()) == g.order()
True
```

Lex DFS ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: g.lex_DFS()
[1, 2, 3, 5, 6, 4]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_DFS(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

sage.graphs.traversals.**lex_DOWN**(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic DOWN search (LexDOWN) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. During the i-th iteration of the algorithm $n - i$ is prepended to the codes of all neighbors of the selected vertex that are left in the graph.

Time complexity is $O(n + m)$ where n is the number of vertices and m is the number of edges.

See [Mil2017] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_UP()* – perform a lexicographic UP search (LexUP) on the graph

EXAMPLES:

A Lex DOWN is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_DOWN()) == g.order()
True
```

Lex DOWN ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: g.lex_DOWN()
[1, 2, 3, 4, 6, 5]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_DOWN(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

sage.graphs.traversals.**lex_M**(*self*, *triangulation=False*, *labels=False*, *initial_vertex=None*, *algorithm=None*)

Return an ordering of the vertices according the LexM graph traversal.

LexM is a lexicographic ordering scheme that is a special type of breadth-first-search. LexM can also produce a triangulation of the given graph. This functionality is implemented in this method. For more details on the algorithms used see Sections 4 (`'lex_M_slow'`) and 5.3 (`'lex_M_fast'`) of [RTL76].

Note: This method works only for undirected graphs.

INPUT:

- `triangulation` – boolean (default: `False`); whether to return a list of edges that need to be added in order to triangulate the graph
- `labels` – boolean (default: `False`); whether to return the labels assigned to each vertex
- `initial_vertex` – (default: `None`); the first vertex to consider
- `algorithm` – string (default: `None`); one of the following algorithms:
  - `'lex_M_slow'`: slower implementation of LexM traversal
  - `'lex_M_fast'`: faster implementation of LexM traversal (works only when `labels` is set to `False`)
  - `None`: Sage chooses the best algorithm: `'lex_M_slow'` if `labels` is set to `True`, `'lex_M_fast'` otherwise.

OUTPUT:

Depending on the values of the parameters `triangulation` and `labels` the method will return one or more of the following (in that order):

- an ordering of vertices of the graph according to LexM ordering scheme
- the labels assigned to each vertex
- a list of edges that when added to the graph will triangulate it

EXAMPLES:

LexM produces an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: ord = g.lex_M(algorithm='lex_M_fast')
sage: len(ord) == g.order()
True
sage: set(ord) == set(g.vertices())
True
sage: ord = g.lex_M(algorithm='lex_M_slow')
sage: len(ord) == g.order()
True
sage: set(ord) == set(g.vertices())
True
```

Both algorithms produce a valid LexM ordering α (i.e the neighbors of $\alpha(i)$ in $G[\{\alpha(i), ..., \alpha(n)\}]$ induce a clique):

```
sage: from sage.graphs.traversals import is_valid_lex_M_order
sage: G = graphs.PetersenGraph()
sage: ord, F = G.lex_M(triangulation=True, algorithm='lex_M_slow')
sage: is_valid_lex_M_order(G, ord, F)
True
sage: ord, F = G.lex_M(triangulation=True, algorithm='lex_M_fast')
sage: is_valid_lex_M_order(G, ord, F)
True
```

LexM produces a triangulation of given graph:

```
sage: G = graphs.PetersenGraph()
sage: _, F = G.lex_M(triangulation=True)
sage: H = Graph(F, format='list_of_edges')
sage: H.is_chordal()
True
```

LexM ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: g.lex_M()
[6, 4, 5, 3, 2, 1]
```

sage.graphs.traversals.**lex_M_fast**(*G*, *triangulation=False*, *initial_vertex=None*)

Return an ordering of the vertices according the LexM graph traversal.

LexM is a lexicographic ordering scheme that is a special type of breadth-first-search. This function implements the algorithm described in Section 5.3 of [RTL76].

Note that instead of using labels $1, 2, \ldots, k$ and adding $1/2$, we use labels $2, 4, \ldots, k$ and add 1, thus avoiding to use floats or rationals.

Note: This method works only for undirected graphs.

INPUT:

- `G` – a sage graph
- `triangulation` – boolean (default: `False`); whether to return the triangulation of given graph produced by the method
- `initial_vertex` – (default: `None`); the first vertex to consider

OUTPUT:

This method will return an ordering of the vertices of `G` according to the LexM ordering scheme. Furthermore, if `triangulation` is set to `True` the method also returns a list of edges `F` such that when added to `G` the resulting graph is a triangulation of `G`.

EXAMPLES:

A LexM ordering is obviously an ordering of the vertices:

```
sage: from sage.graphs.traversals import lex_M_fast
sage: g = graphs.CompleteGraph(6)
sage: len(lex_M_fast(g)) == g.order()
True
```

LexM ordering of the 3-sun graph:

```
sage: from sage.graphs.traversals import lex_M_fast
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: lex_M_fast(g)
[6, 4, 5, 3, 2, 1]
```

LexM produces a triangulation of given graph:

```
sage: from sage.graphs.traversals import lex_M_fast
sage: G = graphs.PetersenGraph()
sage: _, F = lex_M_fast(G, triangulation=True)
sage: H = G.copy()
sage: H.add_edges(F)
sage: H.is_chordal()
True
```

sage.graphs.traversals.**lex_M_slow**(*G*, *triangulation=False*, *labels=False*, *initial_vertex=None*)

Return an ordering of the vertices according the LexM graph traversal.

LexM is a lexicographic ordering scheme that is a special type of breadth-first-search. This function implements the algorithm described in Section 4 of [RTL76].

During the search, the vertices are numbered from n to 1. Let $\alpha(i)$ denote the vertex numbered i and let $\alpha^{-1}(u)$ denote the number assigned to u. Each vertex u has also a label, denoted by $label(u)$, consisting of a list of numbers selected from $[1, n]$ and ordered in decreasing order. Given two labels $L_1 = [p_1, p_2, \ldots, p_k]$ and $L_1 = [q_1, q_2, \ldots, q_l]$, we define $L_1 < L_2$ if, for some j, $p_i == q_i$ for $i = 1, \ldots, j-1$ and $p_j < q_j$, or if $p_i == q_i$ for $i = 1, \ldots, k$ and $k < l$. Observe that this is exactly how Python compares two lists.

Note: This method works only for undirected graphs.

INPUT:

- `G` – a sage graph
- `triangulation` – boolean (default: `False`); whether to return the triangulation of the graph produced by the method
- `labels` – boolean (default: `False`); whether to return the labels assigned to each vertex
- `initial_vertex` – (default: `None`); the first vertex to consider. If not specified, an arbitrary vertex is chosen.

OUTPUT:

Depending on the values of the parameters `triangulation` and `labels` the method will return one or more of the following (in that order):

- the ordering of vertices of G
- the labels assigned to each vertex
- a list of edges that when added to G will produce a triangulation of G

EXAMPLES:

A LexM ordering is obviously an ordering of the vertices:

```
sage: from sage.graphs.traversals import lex_M_slow
sage: g = graphs.CompleteGraph(6)
sage: len(lex_M_slow(g)) == g.order()
True
```

LexM ordering and label assignments on the vertices of the 3-sun graph:

```
sage: from sage.graphs.traversals import lex_M_slow
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: lex_M_slow(g, labels=True)
([6, 4, 5, 3, 2, 1],
 {1: [], 2: [5], 3: [5, 4], 4: [4, 2], 5: [4, 3], 6: [3, 2]})
```

LexM produces a triangulation of given graph:

```
sage: from sage.graphs.traversals import lex_M_slow
sage: G = graphs.PetersenGraph()
sage: _, F = lex_M_slow(G, triangulation=True)
sage: H = G.copy()
sage: H.add_edges(F)
sage: H.is_chordal()
True
```

`sage.graphs.traversals.`**`lex_UP`**(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Perform a lexicographic UP search (LexUP) on the graph.

INPUT:

- `G` – a sage graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse

- `tree` – boolean (default: `False`); whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

ALGORITHM:

This algorithm maintains for each vertex left in the graph a code corresponding to the vertices already removed. The vertex of maximal code (according to the lexicographic order) is then removed, and the codes are updated. During the i-th iteration of the algorithm i is appended to the codes of all neighbors of the selected vertex that are left in the graph.

Time complexity is $O(n+m)$ where n is the number of vertices and m is the number of edges.

See [Mil2017] for more details on the algorithm.

See also:

- *lex_BFS()* – perform a lexicographic breadth first search (LexBFS) on the graph
- *lex_DFS()* – perform a lexicographic depth first search (LexDFS) on the graph
- *lex_DOWN()* – perform a lexicographic DOWN search (LexDOWN) on the graph

EXAMPLES:

A Lex UP is obviously an ordering of the vertices:

```
sage: g = graphs.CompleteGraph(6)
sage: len(g.lex_UP()) == g.order()
True
```

Lex UP ordering of the 3-sun graph:

```
sage: g = Graph([(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 5),␣
↪(5, 6)])
sage: g.lex_UP()
[1, 2, 4, 5, 6, 3]
```

The method also works for directed graphs:

```
sage: G = DiGraph([(1, 2), (2, 3), (1, 3)])
sage: G.lex_UP(initial_vertex=2)
[2, 3, 1]
```

Different orderings for different traversals:

```
sage: G = digraphs.DeBruijn(2,3)
sage: G.lex_BFS(initial_vertex='000')
['000', '001', '100', '010', '011', '110', '101', '111']
sage: G.lex_DFS(initial_vertex='000')
['000', '001', '100', '010', '101', '110', '011', '111']
sage: G.lex_UP(initial_vertex='000')
['000', '001', '010', '101', '110', '111', '011', '100']
sage: G.lex_DOWN(initial_vertex='000')
['000', '001', '100', '011', '010', '110', '111', '101']
```

sage.graphs.traversals.**maximum_cardinality_search**(*G*, *reverse=False*, *tree=False*, *initial_vertex=None*)

Return an ordering of the vertices according a maximum cardinality search.

Maximum cardinality search (MCS) is a graph traversal introduced in [TY1984]. It starts by assigning an arbitrary vertex (or the specified `initial_vertex`) of G the last position in the ordering α. Every vertex keeps a weight equal to the number of its already processed neighbors (i.e., already added to α), and a vertex of largest such number is chosen at each step i to be placed in position $n - i$ in α. This ordering can be computed in time $O(n + m)$.

When the graph is chordal, the ordering returned by MCS is a *perfect elimination ordering*, like *lex_BFS()*. So this ordering can be used to recognize chordal graphs. See [He2006] for more details.

Note: The current implementation is for connected graphs only.

INPUT:

- `G` – a Sage Graph
- `reverse` – boolean (default: `False`); whether to return the vertices in discovery order, or the reverse
- `tree` – boolean (default: `False`); whether to also return the discovery directed tree (each vertex being linked to the one that saw it for the first time)
- `initial_vertex` – (default: `None`); the first vertex to consider

OUTPUT:

By default, return the ordering α as a list. When `tree` is `True`, the method returns a tuple (α, T), where T is a directed tree with the same set of vertices as G to v if u was the first vertex to saw v.

EXAMPLES:

When specified, the `initial_vertex` is placed at the end of the ordering, unless parameter `reverse` is `True`, in which case it is placed at the beginning:

```
sage: G = graphs.PathGraph(4)
sage: G.maximum_cardinality_search(initial_vertex=0)
[3, 2, 1, 0]
sage: G.maximum_cardinality_search(initial_vertex=1)
[0, 3, 2, 1]
sage: G.maximum_cardinality_search(initial_vertex=2)
[0, 1, 3, 2]
sage: G.maximum_cardinality_search(initial_vertex=3)
[0, 1, 2, 3]
sage: G.maximum_cardinality_search(initial_vertex=3, reverse=True)
[3, 2, 1, 0]
```

Returning the discovery tree:

```
sage: G = graphs.PathGraph(4)
sage: _, T = G.maximum_cardinality_search(tree=True, initial_vertex=0)
sage: T.order(), T.size()
(4, 3)
sage: T.edges(labels=False, sort=True)
[(1, 0), (2, 1), (3, 2)]
sage: _, T = G.maximum_cardinality_search(tree=True, initial_vertex=3)
sage: T.edges(labels=False, sort=True)
[(0, 1), (1, 2), (2, 3)]
```

sage.graphs.traversals.**maximum_cardinality_search_M**(*G*, *initial_vertex=None*)

Return the ordering and the edges of the triangulation produced by MCS-M.

Maximum cardinality search M (MCS-M) is an extension of MCS (*maximum_cardinality_search()*) in the same way that Lex-M (*lex_M()*) is an extension of Lex-BFS (*lex_BFS()*). That is, in MCS-M when u receives number i at step $n - i + 1$, it increments the weight of all unnumbered vertices v for which there exists a path between u and v consisting only of unnumbered vertices with weight strictly less than $w^-(u)$ and $w^-(v)$, where w^- is the number of times a vertex has been reached during previous iterations. See [BBHP2004] for the details of this $O(nm)$ time algorithm.

If G is not connected, the orderings of each of its connected components are added consecutively. Furthermore, if G has k connected components C_i for $0 \leq i < k$, X contains at least one vertex of C_i for each $i \geq 1$. Hence, $|X| \geq k - 1$. In particular, some isolated vertices (i.e., of degree 0) can appear in X as for such a vertex x, we have that $G \setminus N(x) = G$ is not connected.

INPUT:

- `G` – a Sage graph
- `initial_vertex` – (default: `None`); the first vertex to consider

OUTPUT: a tuple (α, F, X), where

- α is the resulting ordering of the vertices. If an initial vertex is specified, it gets the last position in the ordering α.
- F is the list of edges of a minimal triangulation of G according α
- X is is a list of vertices such that for each $x \in X$, the neighborhood of x in G is a separator (i.e., $G \setminus N(x)$ is not connected). Note that we may have $N(x) = \emptyset$ if G is not connected and x has degree 0.

EXAMPLES:

Chordal graphs have a perfect elimination ordering, and so the set F of edges of the triangulation is empty:

```
sage: G = graphs.RandomChordalGraph(20)
sage: alpha, F, X = G.maximum_cardinality_search_M(); F
[]
```

The cycle of order 4 is not chordal and so the triangulation has one edge:

```
sage: G = graphs.CycleGraph(4)
sage: alpha, F, X = G.maximum_cardinality_search_M(); len(F)
1
```

The number of edges needed to triangulate of a cycle graph or order n is $n - 3$, independently of the initial vertex:

```
sage: n = randint(3, 20)
sage: C = graphs.CycleGraph(n)
sage: _, F, X = C.maximum_cardinality_search_M()
sage: len(F) == n - 3
True
sage: _, F, X = C.maximum_cardinality_search_M(initial_vertex=C.random_vertex())
sage: len(F) == n - 3
True
```

When an initial vertex is specified, it gets the last position in the ordering:

```
sage: G = graphs.PathGraph(4)
sage: G.maximum_cardinality_search_M(initial_vertex=0)
([3, 2, 1, 0], [], [2, 3])
sage: G.maximum_cardinality_search_M(initial_vertex=1)
```

(continues on next page)

(continued from previous page)

```
([3, 2, 0, 1], [], [2, 3])
sage: G.maximum_cardinality_search_M(initial_vertex=2)
([0, 1, 3, 2], [], [0, 1])
sage: G.maximum_cardinality_search_M(initial_vertex=3)
([0, 1, 2, 3], [], [0, 1])
```

When G is not connected, the orderings of each of its connected components are added consecutively, the vertices of the component containing the initial vertex occupying the last positions:

```
sage: G = graphs.CycleGraph(4) * 2
sage: G.maximum_cardinality_search_M()[0]
[5, 4, 6, 7, 2, 3, 1, 0]
sage: G.maximum_cardinality_search_M(initial_vertex=7)[0]
[2, 1, 3, 0, 5, 6, 4, 7]
```

Furthermore, if G has k connected components, X contains at least one vertex per connected component, except for the first one, and so at least $k-1$ vertices:

```
sage: for k in range(1, 5):
....:     _, _, X = Graph(k).maximum_cardinality_search_M()
....:     if len(X) < k - 1:
....:         raise ValueError("something goes wrong")
sage: G = graphs.RandomGNP(10, .2)
sage: cc = G.connected_components()
sage: _, _, X = G.maximum_cardinality_search_M()
sage: len(X) >= len(cc) - 1
True
```

In the example of [BPS2010], the triangulation has 3 edges:

```
sage: G = Graph({'a': ['b', 'k'], 'b': ['c'], 'c': ['d', 'j', 'k'],
....:            'd': ['e', 'f', 'j', 'k'], 'e': ['g'],
....:            'f': ['g', 'j', 'k'], 'g': ['j', 'k'], 'h': ['i', 'j'],
....:            'i': ['k'], 'j': ['k']})
sage: _, F, _ = G.maximum_cardinality_search_M(initial_vertex='a')
sage: len(F)
3
```

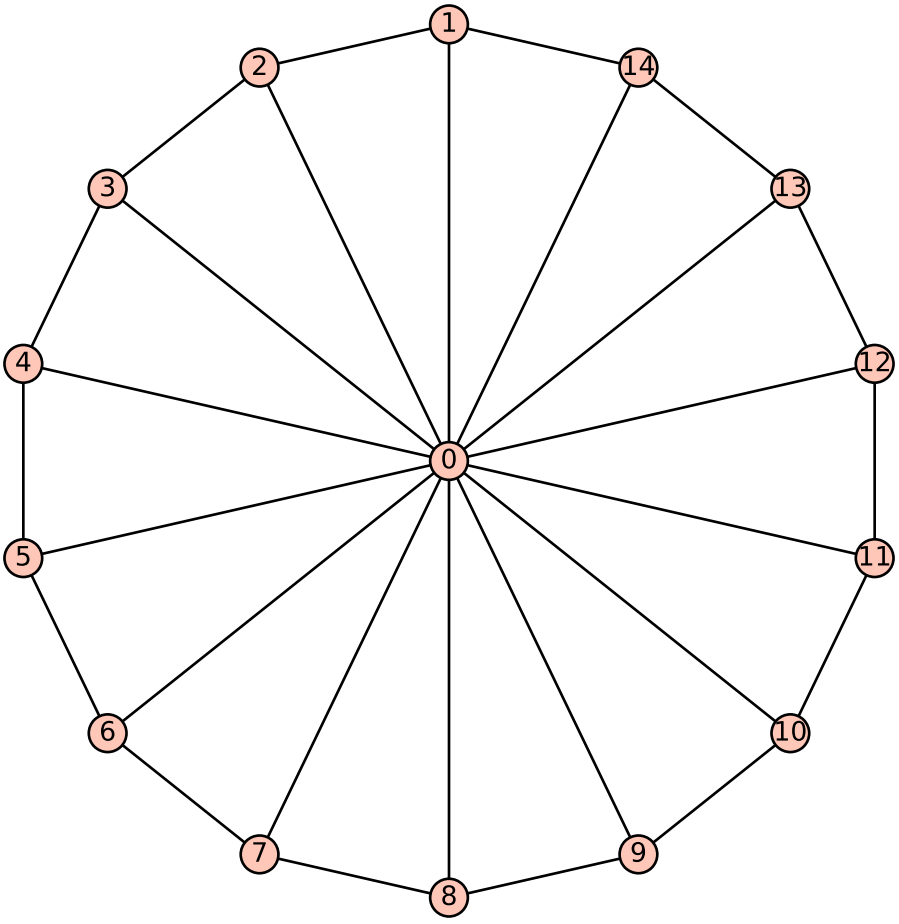
5.17 Graph Plotting

(For LaTeX drawings of graphs, see the `graph_latex` *module.)*

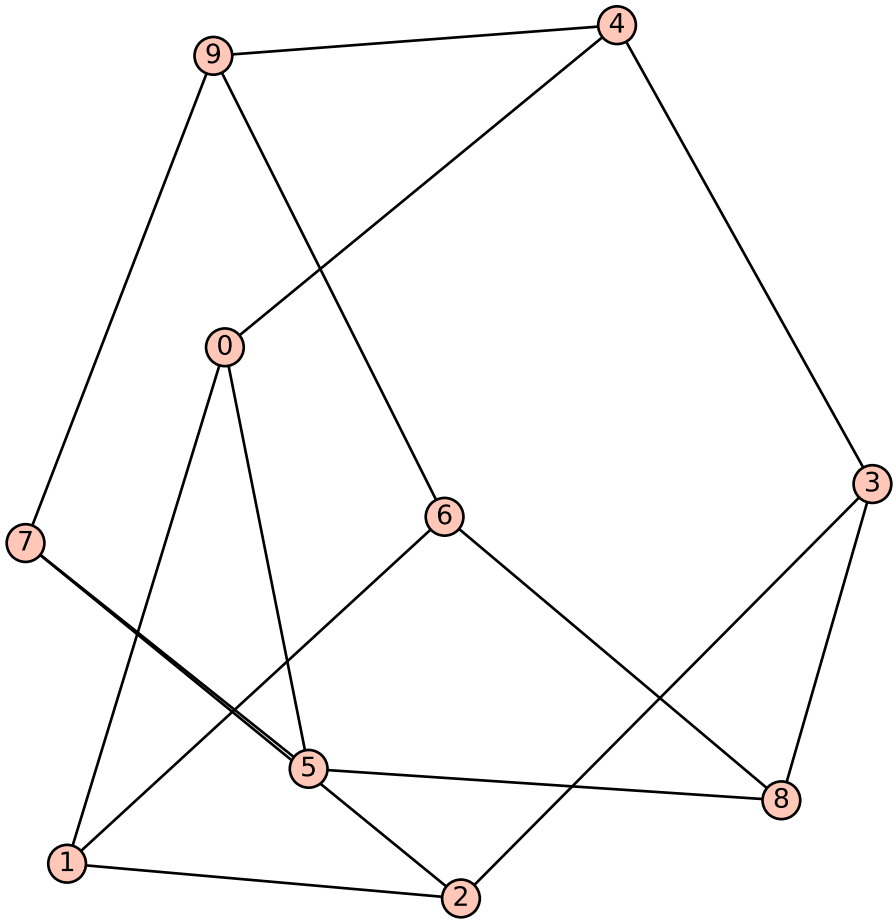
All graphs have an associated Sage graphics object, which you can display:

```
sage: G = graphs.WheelGraph(15)
sage: P = G.plot()
sage: P.show() # long time
```

If you create a graph in Sage using the `Graph` command, then plot that graph, the positioning of nodes is determined using the spring-layout algorithm. For the special graph constructors, which you get using `graphs.[tab]`, the positions are preset. For example, consider the Petersen graph with default node positioning vs. the Petersen graph constructed by this database:


1
2
14
3
13
4
12
0
5
11
6
10
7
9
8

```
sage: petersen_spring = Graph(':I`ES@obGkqegW~')
sage: petersen_spring.show() # long time
```

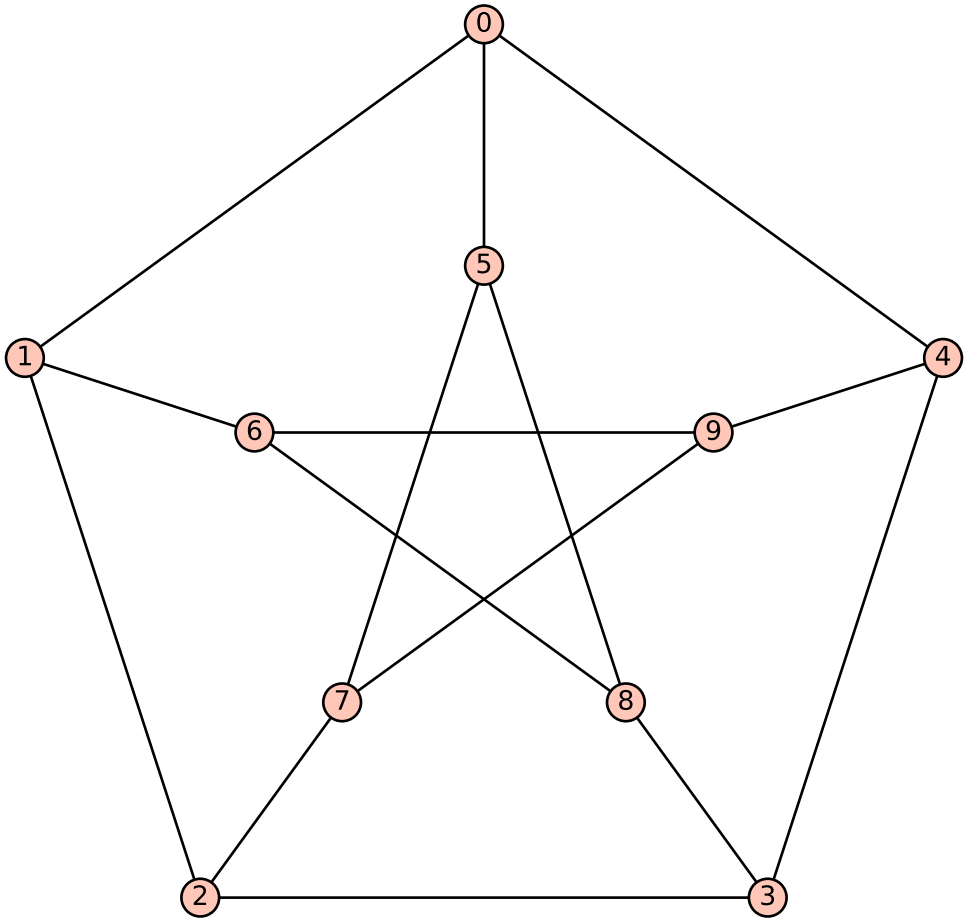



```
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

For all the constructors in this database (except some random graphs), the position dictionary is filled in, instead of using the spring-layout algorithm.

Plot options

Here is the list of options accepted by *plot()* and the constructor of *GraphPlot*. Those two functions also accept all options of sage.plot.graphics.Graphics.show().


0
1
2
3
4
5
6
7
8
9

layout	A layout algorithm – one of : "acyclic", "circular" (plots the graph with vertices evenly distributed on a circle), "ranked", "graphviz", "planar", "spring" (traditional spring layout, using the graph's current positions as initial positions), or "tree" (the tree will be plotted in levels, depending on minimum distance for the root).
iterations	The number of times to execute the spring layout algorithm.
heights	A dictionary mapping heights to the list of vertices at this height.
spring	Use spring layout to finalize the current layout.
tree_root	A vertex designation for drawing trees. A vertex of the tree to be used as the root for the layout='tree' option. If no root is specified, then one is chosen close to the center of the tree. Ignored unless layout='tree'
tree_orientation	The direction of tree branches – 'up', 'down', 'left' or 'right'.
save_pos	Whether or not to save the computed position for the graph.
dim	The dimension of the layout – 2 or 3.
prog	Which graphviz layout program to use – one of "circo", "dot", "fdp", "neato", or "twopi".
by_component	Whether to do the spring layout by connected component – a boolean.
pos	The position dictionary of vertices
vertex_labels	Whether or not to draw vertex labels.
vertex_color	Default color for vertices not listed in vertex_colors dictionary.
vertex_colors	Dictionary of vertex coloring : each key is a color recognizable by matplotlib, and each corresponding entry is a list of vertices.
vertex_size	The size to draw the vertices.
vertex_shape	The shape to draw the vertices. Currently unavailable for Multi-edged DiGraphs.
edge_labels	Whether or not to draw edge labels.
edge_style	The linestyle of the edges. It should be one of "solid", "dashed", "dotted", dashdot", or "-", "–", ":", "-.", respectively.
edge_thickness	The thickness of the edges.
edge_color	The default color for edges not listed in edge_colors.
edge_colors	a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges.
color_by_label	Whether to color the edges according to their labels. This also accepts a function or dictionary mapping labels to colors.
partition	A partition of the vertex set. If specified, plot will show each cell in a different color. vertex_colors takes precedence.
loop_size	The radius of the smallest loop.
dist	The distance between multiedges.
max_dist	The max distance range to allow multiedges.
talk	Whether to display the vertices in talk mode (larger and white).
graph_border	Whether or not to draw a frame around the graph.
edge_labels_background	The color of the background of the edge labels

Default options

This module defines two dictionaries containing default options for the *plot()* and *show()* methods. These two dictionaries are sage.graphs.graph_plot.DEFAULT_PLOT_OPTIONS and sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS, respectively.

Obviously, these values are overruled when arguments are given explicitly.

Here is how to define the default size of a graph drawing to be [6,6]. The first two calls to *show()* use this option, while the third does not (a value for figsize is explicitly given):

```
sage: import sage.graphs.graph_plot
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [6,6]
sage: graphs.PetersenGraph().show() # long time
sage: graphs.ChvatalGraph().show()  # long time
sage: graphs.PetersenGraph().show(figsize=[4,4]) # long time
```

We can now reset the default to its initial value, and now display graphs as previously:

```
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [4,4]
sage: graphs.PetersenGraph().show() # long time
sage: graphs.ChvatalGraph().show()  # long time
```

Note:

- While `DEFAULT_PLOT_OPTIONS` affects both `G.show()` and `G.plot()`, settings from `DEFAULT_SHOW_OPTIONS` only affects `G.show()`.
- In order to define a default value permanently, you can add a couple of lines to Sage's startup scripts. Example:

```
sage: import sage.graphs.graph_plot
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [4,4]
```

Index of methods and functions

GraphPlot.set_pos()	Set the position plotting parameters for this GraphPlot.
GraphPlot.set_vertices()	Set the vertex plotting parameters for this GraphPlot.
GraphPlot.set_edges()	Set the edge (or arrow) plotting parameters for the GraphPlot object.
GraphPlot.show()	Show the (Di)Graph associated with this GraphPlot object.
GraphPlot.plot()	Return a graphics object representing the (di)graph.
GraphPlot.layout_tree()	Compute a nice layout of a tree.

class sage.graphs.graph_plot.**GraphPlot**(*graph*, *options*)

Bases: sage.structure.sage_object.SageObject

Return a `GraphPlot` object, which stores all the parameters needed for plotting (Di)Graphs.

A `GraphPlot` has a plot and show function, as well as some functions to set parameters for vertices and edges. This constructor assumes default options are set. Defaults are shown in the example below.

EXAMPLES:

```
sage: from sage.graphs.graph_plot import GraphPlot
sage: options = {
....:   'vertex_size': 200,
....:   'vertex_labels': True,
....:   'layout': None,
....:   'edge_style': 'solid',
....:   'edge_color': 'black',
....:   'edge_colors': None,
....:   'edge_labels': False,
....:   'iterations': 50,
....:   'tree_orientation': 'down',
....:   'heights': None,
```

(continues on next page)

(continued from previous page)

```
....:    'graph_border': False,
....:    'talk': False,
....:    'color_by_label': False,
....:    'partition': None,
....:    'dist': .075,
....:    'max_dist': 1.5,
....:    'loop_size': .075,
....:    'edge_labels_background': 'transparent'}
sage: g = Graph({0:[1, 2], 2:[3], 4:[0, 1]})
sage: GP = GraphPlot(g, options)
```

layout_tree(*root*, *orientation*)

Compute a nice layout of a tree.

INPUT:

- `root` – the root vertex.
- `orientation` – whether to place the root at the top or at the bottom:
  - `orientation="down"` – children are placed below their parent
  - `orientation="top"` – children are placed above their parent

EXAMPLES:

```
sage: from sage.graphs.graph_plot import GraphPlot
sage: G = graphs.HoffmanSingletonGraph()
sage: T = Graph()
sage: T.add_edges(G.min_spanning_tree(starting_vertex=0))
sage: T.show(layout='tree', tree_root=0) # indirect doctest
```

plot(***kwds*)

Return a graphics object representing the (di)graph.

INPUT:

The options accepted by this method are to be found in the documentation of the *sage.graphs.graph_plot* module, and the `show()` method.

Note: See *the module's documentation* for information on default values of this method.

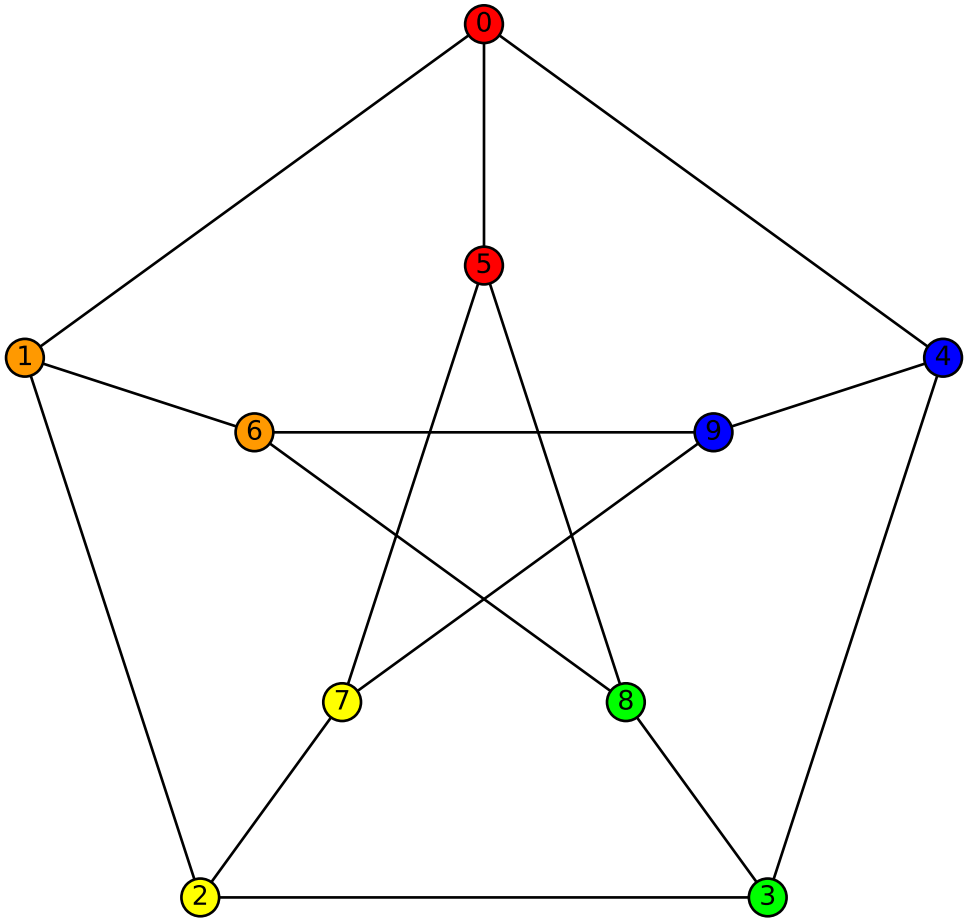
We can specify some pretty precise plotting of familiar graphs:

```
sage: from math import sin, cos, pi
sage: P = graphs.PetersenGraph()
sage: d = {'#FF0000':[0,5], '#FF9900':[1,6], '#FFFF00':[2,7], '#00FF00':[3,8],
....:  '#0000FF':[4,9]}
sage: pos_dict = {}
sage: for i in range(5):
....:  x = float(cos(pi/2 + ((2*pi)/5)*i))
....:  y = float(sin(pi/2 + ((2*pi)/5)*i))
....:  pos_dict[i] = [x,y]
...
sage: for i in range(5,10):
....:  x = float(0.5*cos(pi/2 + ((2*pi)/5)*i))
....:  y = float(0.5*sin(pi/2 + ((2*pi)/5)*i))
....:  pos_dict[i] = [x,y]
```

(continues on next page)

(continued from previous page)

```
...
sage: pl = P.graphplot(pos=pos_dict, vertex_colors=d)
sage: pl.show()
```



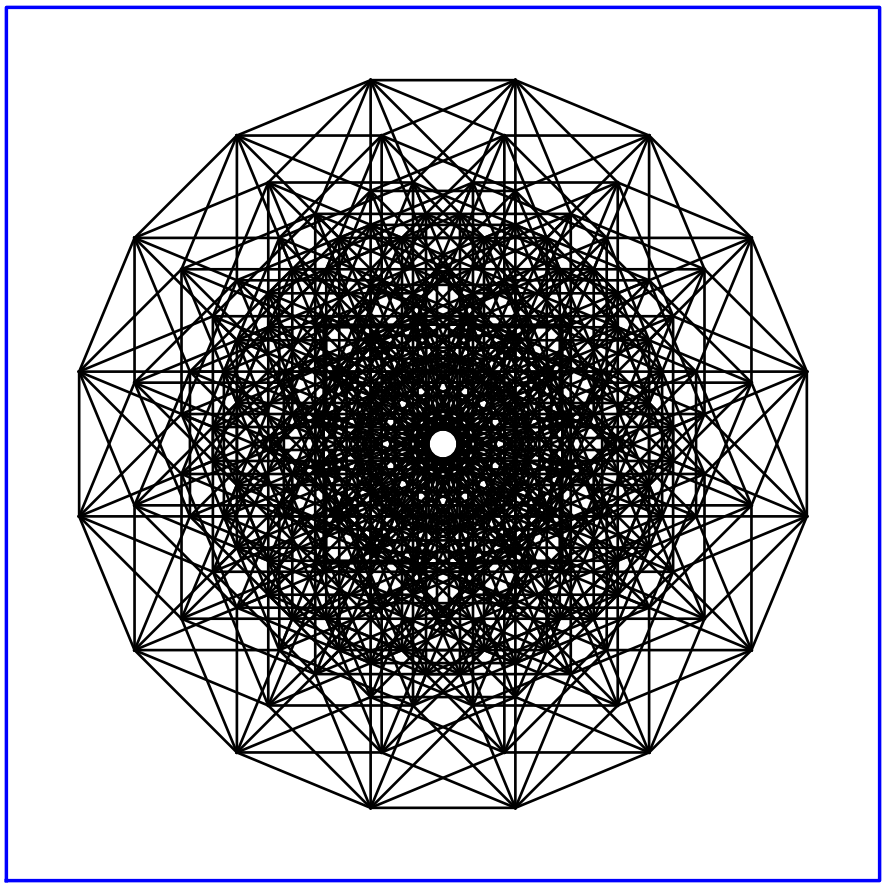
Here are some more common graphs with typical options:

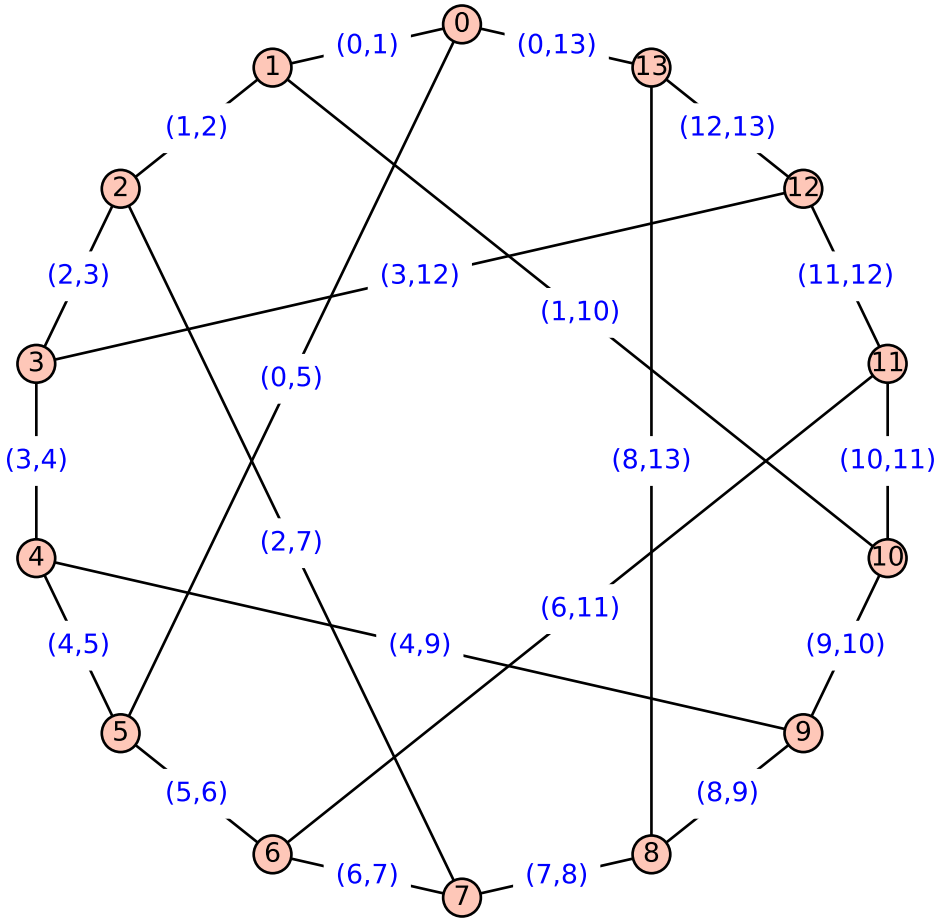
```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

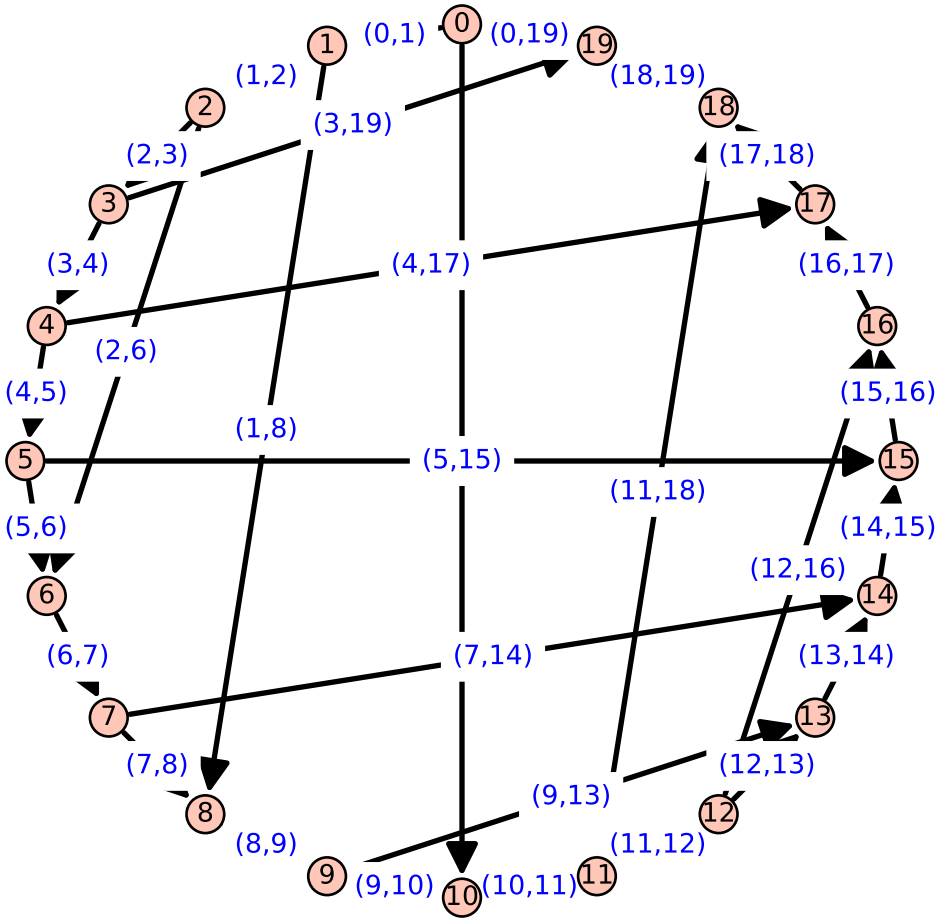
```
sage: G = graphs.HeawoodGraph().copy(sparse=True)
sage: for u,v,l in G.edges():
....:  G.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: G.graphplot(edge_labels=True).show()
```

The options for plotting also work with directed graphs:

```
sage: D = DiGraph( { 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4],
....:  4: [17, 5], 5: [6, 15], 6: [7], 7: [8, 14], 8: [9], 9: [10, 13],
....:  10: [11], 11: [12, 18], 12: [16, 13], 13: [14], 14: [15], 15: [16],
....:  16: [17], 17: [18], 18: [19], 19: []})
sage: for u,v,l in D.edges():
....:  D.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: D.graphplot(edge_labels=True, layout='circular').show()
```

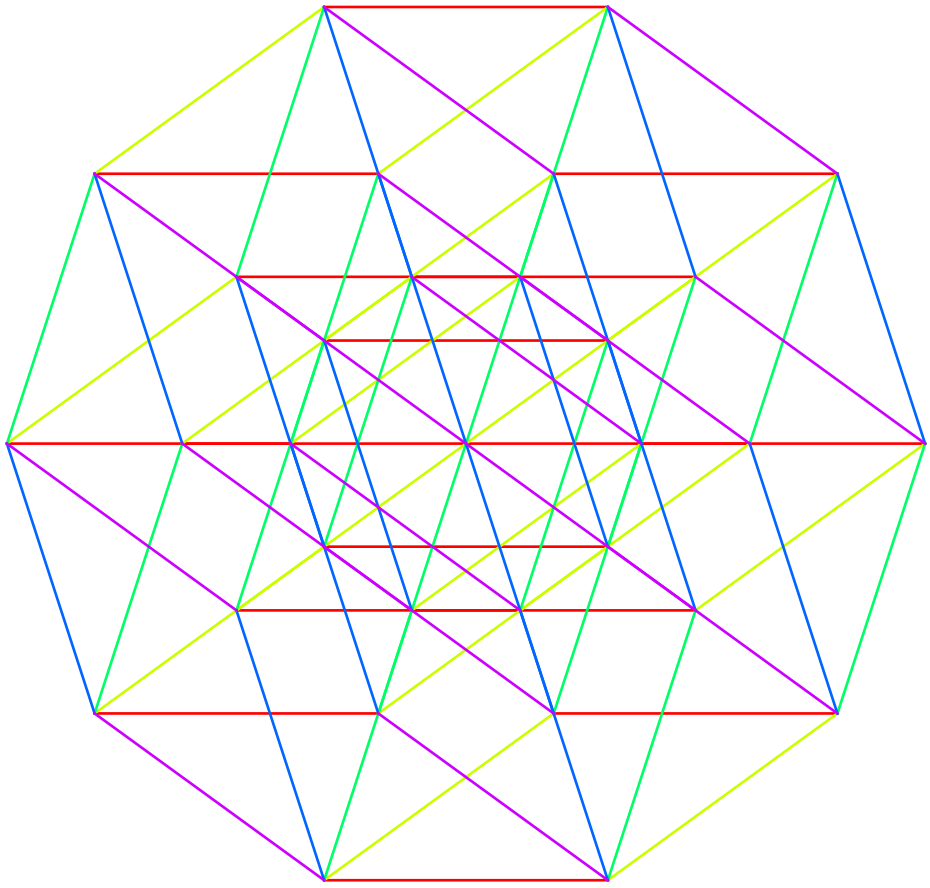



0
1
2
3
4
5
6
7
8
9
10
11
12
13
(0,1)
(0,13)
(1,2)
(12,13)
(2,3)
(3,12)
(11,12)
(1,10)
(0,5)
(3,4)
(8,13)
(10,11)
(2,7)
(6,11)
(4,5)
(4,9)
(9,10)
(5,6)
(8,9)
(6,7)
(7,8)


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
(0,1)
(0,19)
(1,2)
(18,19)
(3,19)
(2,3)
(17,18)
(3,4)
(4,17)
(16,17)
(2,6)
(4,5)
(15,16)
(1,8)
(5,15)
(11,18)
(5,6)
(14,15)
(12,16)
(6,7)
(7,14)
(13,14)
(7,8)
(12,13)
(9,13)
(8,9)
(11,12)
(9,10)
(10,11)

This example shows off the coloring of edges:

```
sage: from sage.plot.colors import rainbow
sage: C = graphs.CubeGraph(5)
sage: R = rainbow(5)
sage: edge_colors = {}
sage: for i in range(5):
....:  edge_colors[R[i]] = []
sage: for u,v,l in C.edges():
....:  for i in range(5):
....:      if u[i] != v[i]:
....:          edge_colors[R[i]].append((u,v,l))
sage: C.graphplot(vertex_labels=False, vertex_size=0, edge_colors=edge_
↪colors).show()
```



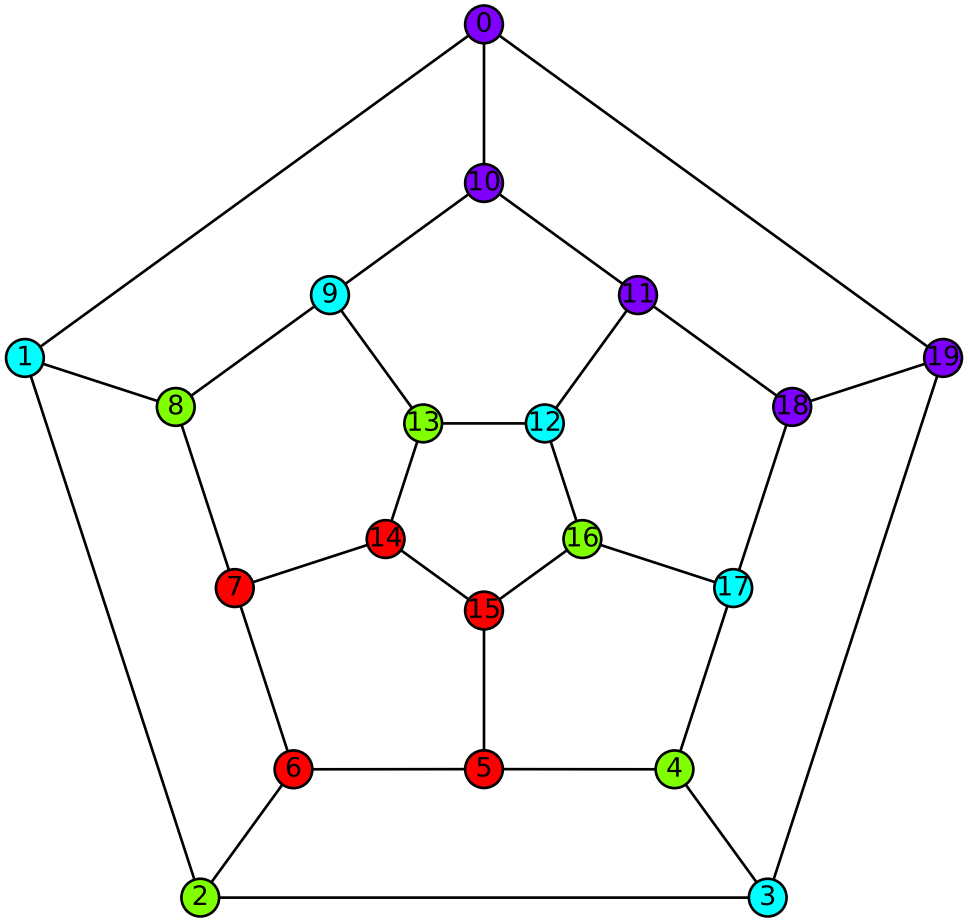
With the `partition` option, we can separate out same-color groups of vertices:

```
sage: D = graphs.DodecahedralGraph()
sage: Pi = [[6,5,15,14,7],[16,13,8,2,4],[12,17,9,3,1],[0,19,18,10,11]]
sage: D.show(partition=Pi)
```

Loops are also plotted correctly:

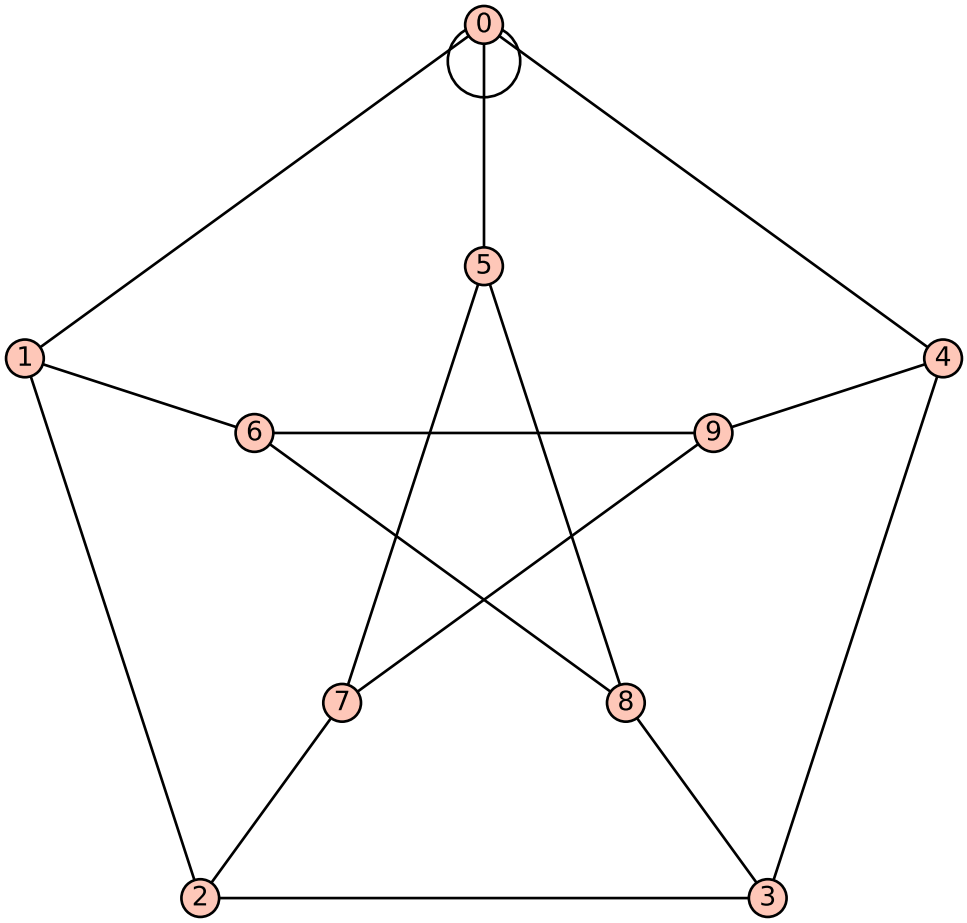
```
sage: G = graphs.PetersenGraph()
sage: G.allow_loops(True)
```

(continues on next page)


0
10
9
11
1
19
8
18
13
12
14
16
7
17
15
6
5
4
2
3

(continued from previous page)

```
sage: G.add_edge(0,0)
sage: G.show()
```



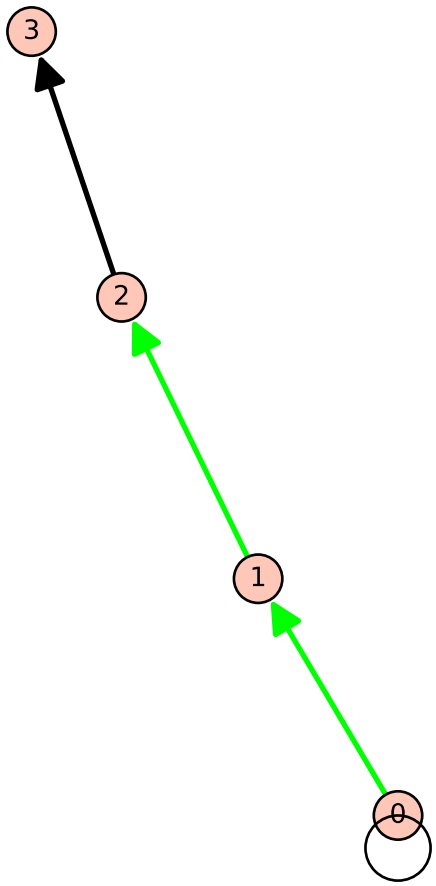

```
sage: D = DiGraph({0:[0,1], 1:[2], 2:[3]}, loops=True)
sage: D.show()
sage: D.show(edge_colors={(0,1,0):[(0,1,None),(1,2,None)],(0,0,0):[(2,3,
↪None)]})
```

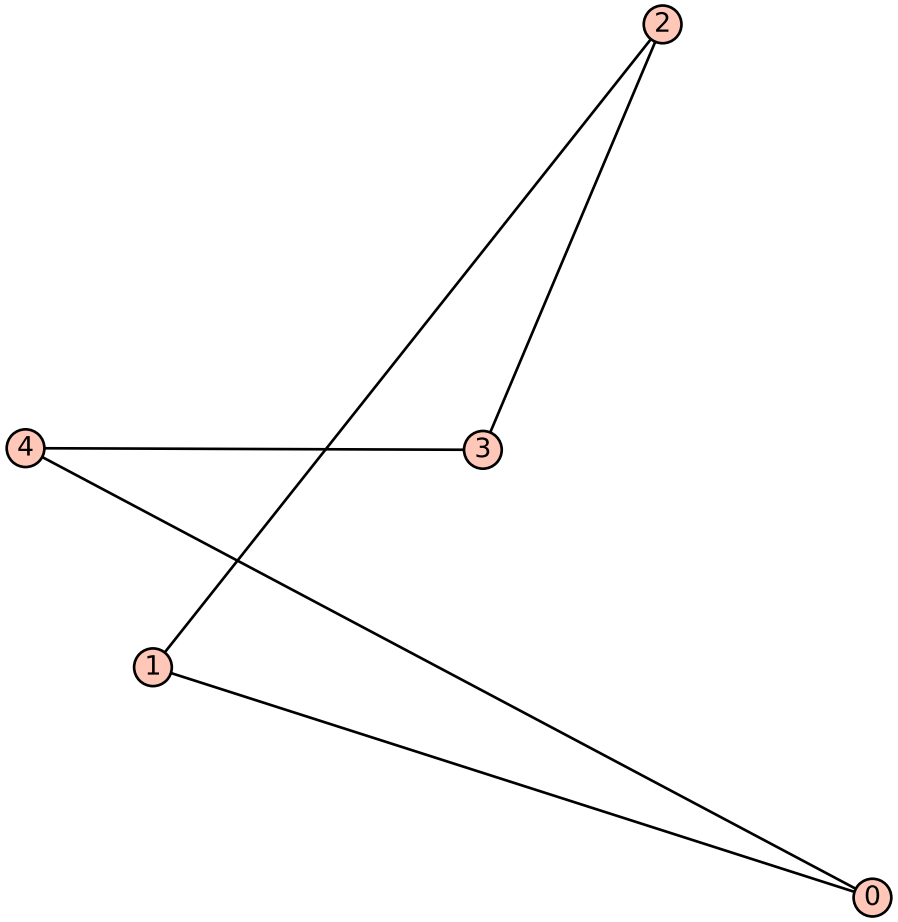
More options:

```
sage: pos = {0:[0.0, 1.5], 1:[-0.8, 0.3], 2:[-0.6, -0.8],
....:  3:[0.6, -0.8], 4:[0.8, 0.3]}
sage: g = Graph({0:[1], 1:[2], 2:[3], 3:[4], 4:[0]})
sage: g.graphplot(pos=pos, layout='spring', iterations=0).plot()
Graphics object consisting of 11 graphics primitives
```

```
sage: G = Graph()
sage: P = G.graphplot().plot()
sage: P.axes()
False
sage: G = DiGraph()
sage: P = G.graphplot().plot()
sage: P.axes()
```

(continues on next page)


3
2
1
0

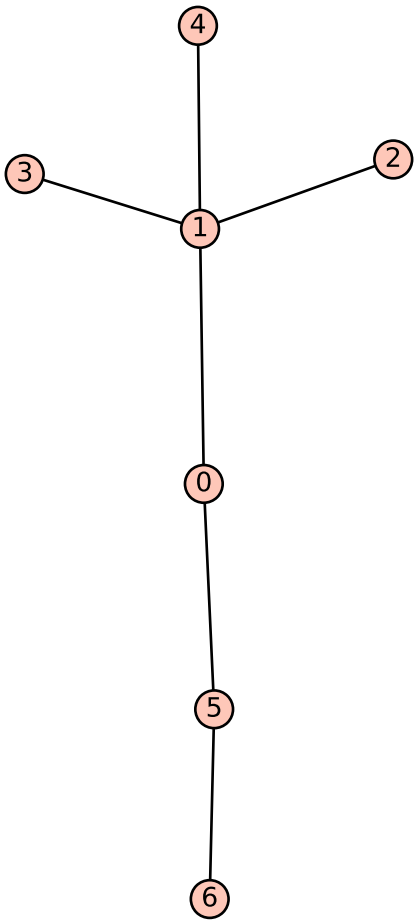

2
4
3
1
0

(continued from previous page)

```
False
```

We can plot multiple graphs:

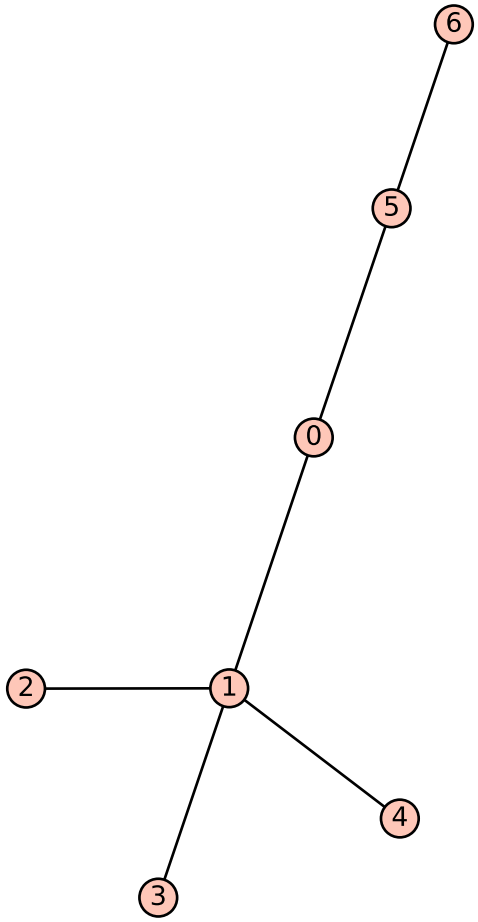
```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}).plot()
Graphics object consisting of 14 graphics primitives
```




```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}).plot()
Graphics object consisting of 14 graphics primitives
```

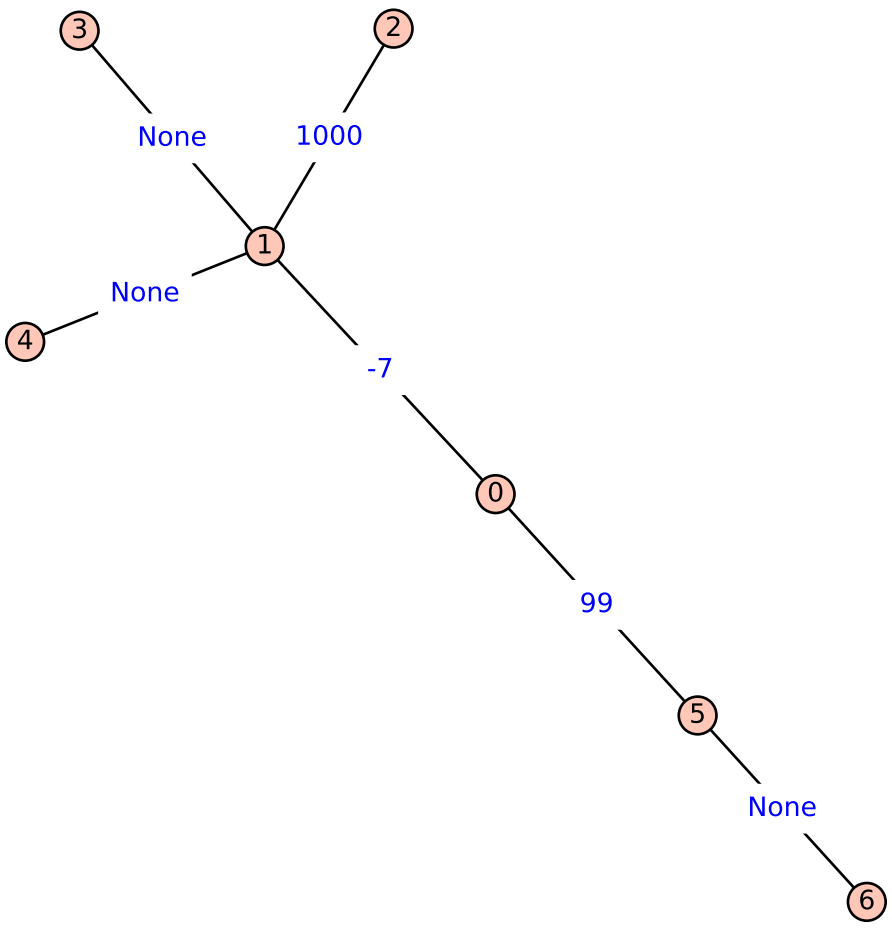
```
sage: t.set_edge_label(0,1,-7)
sage: t.set_edge_label(0,5,3)
sage: t.set_edge_label(0,5,99)
sage: t.set_edge_label(1,2,1000)
sage: t.set_edge_label(3,2,'spam')
sage: t.set_edge_label(2,6,3/2)
sage: t.set_edge_label(0,4,66)
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}, edge_
↪labels=True).plot()
```

(continues on next page)


6
5
0
2
1
4
3

(continued from previous page)

```
Graphics object consisting of 20 graphics primitives
```




```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(layout='tree').show()
```

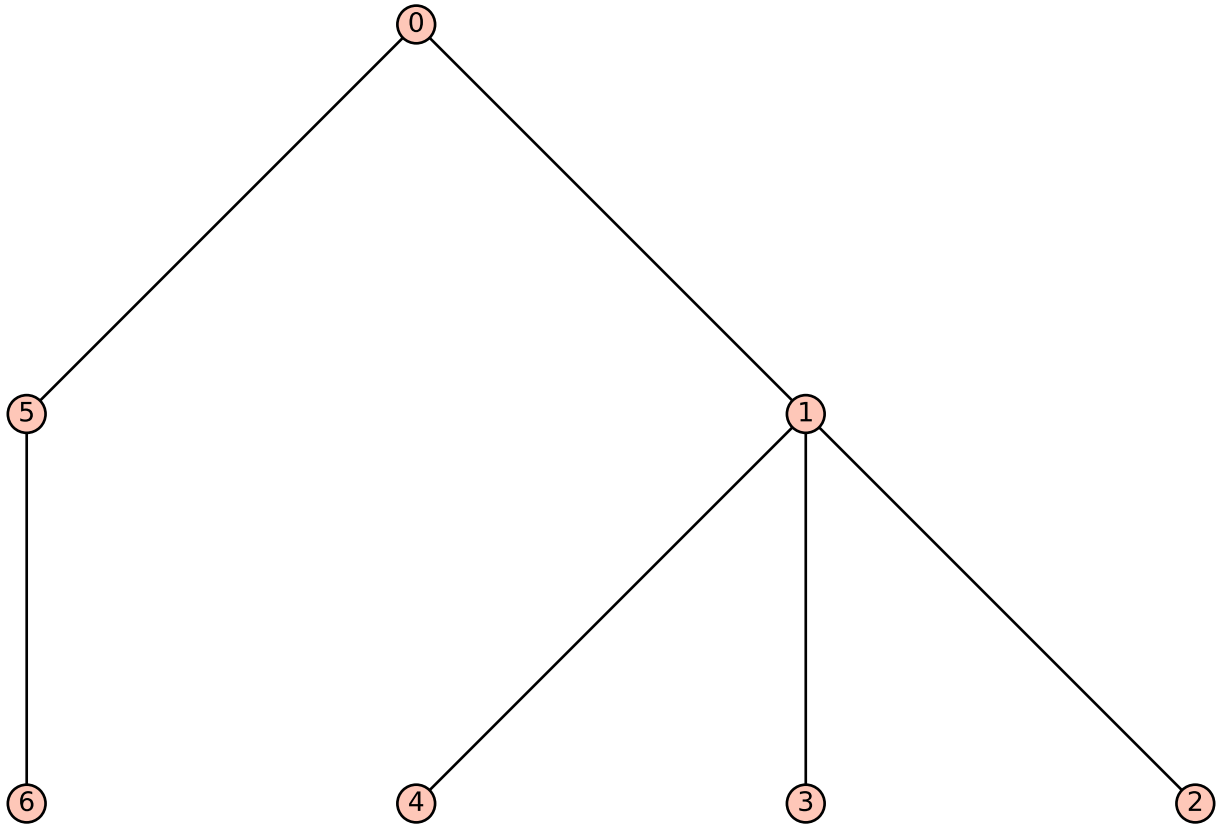
The tree layout is also useful:

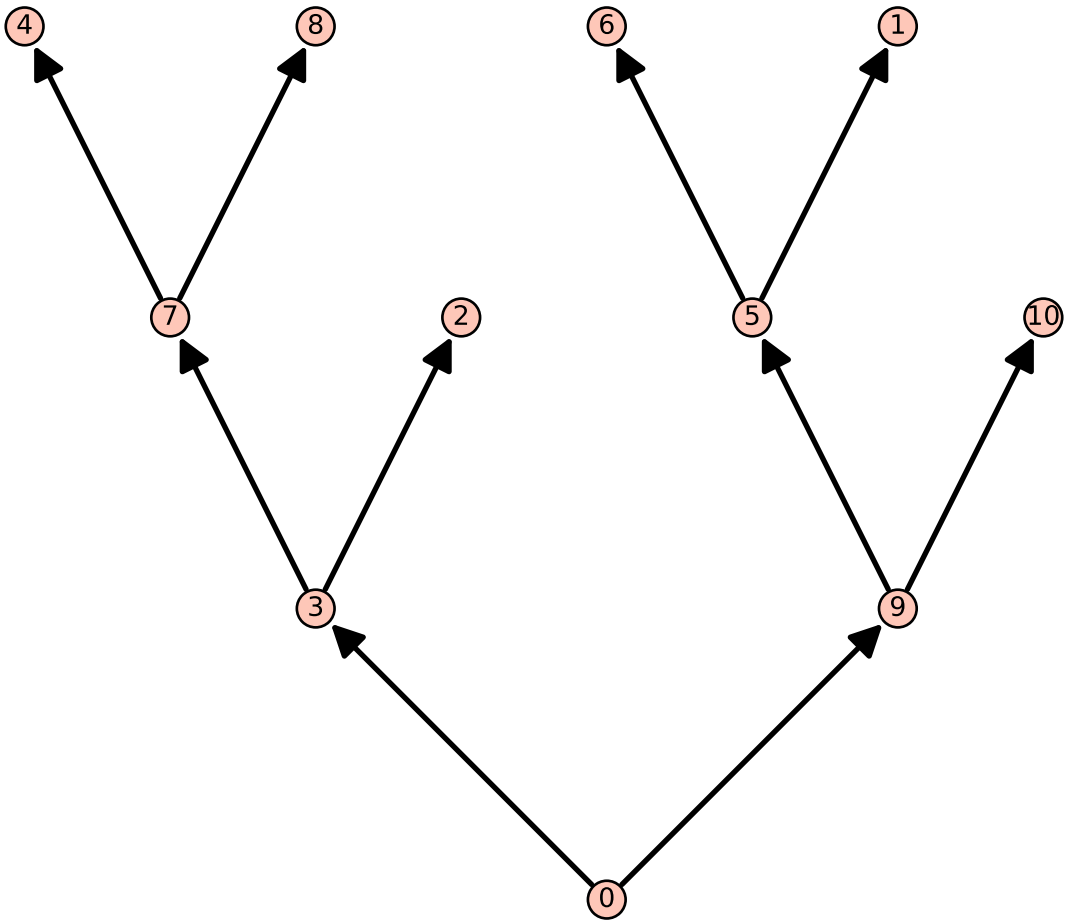
```
sage: t = DiGraph('JCC???@A??GO??CO??GO??')
sage: t.graphplot(layout='tree', tree_root=0, tree_orientation="up").show()
```

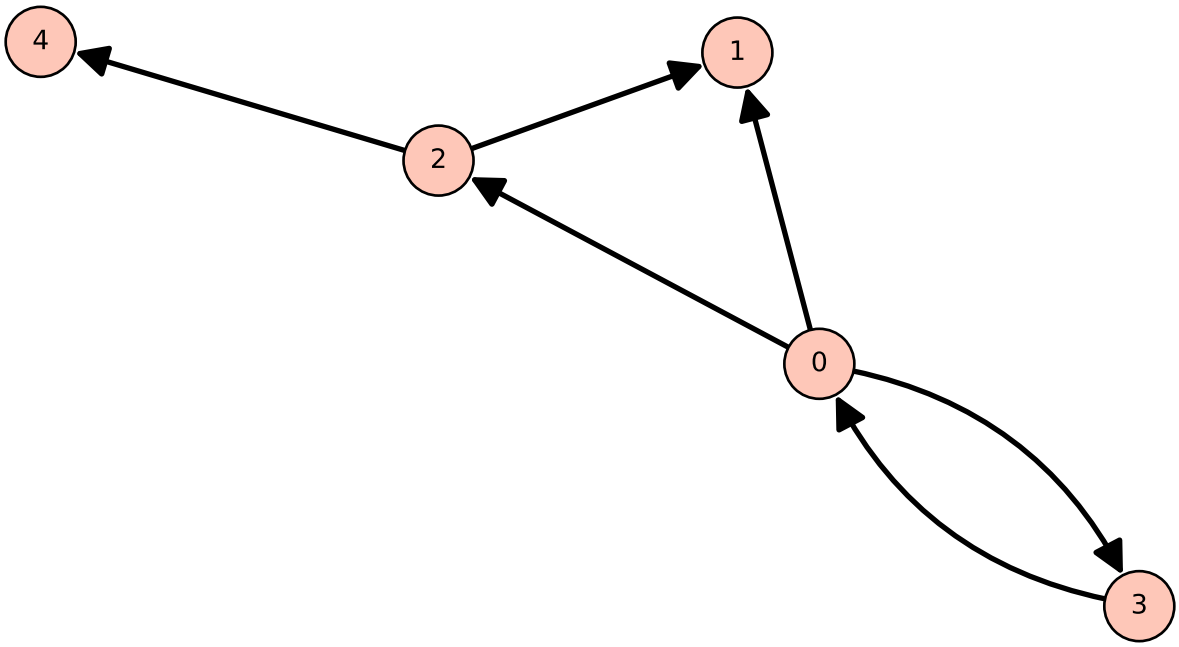
More examples:

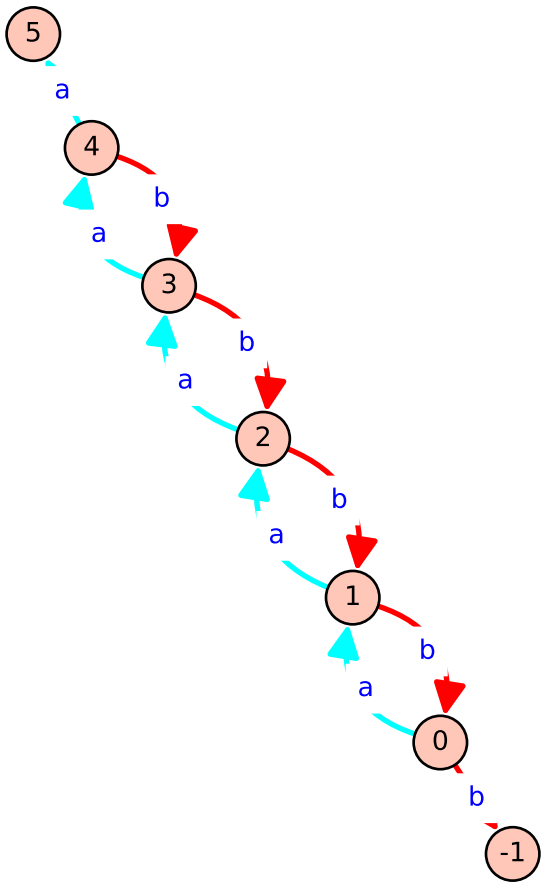
```
sage: D = DiGraph({0:[1,2,3], 2:[1,4], 3:[0]})
sage: D.graphplot().show()
```

```
sage: D = DiGraph(multiedges=True, sparse=True)
sage: for i in range(5):
....:     D.add_edge((i,i+1,'a'))
....:     D.add_edge((i,i-1,'b'))
sage: D.graphplot(edge_labels=True,edge_colors=D._color_by_label()).plot()
Graphics object consisting of 34 graphics primitives
```

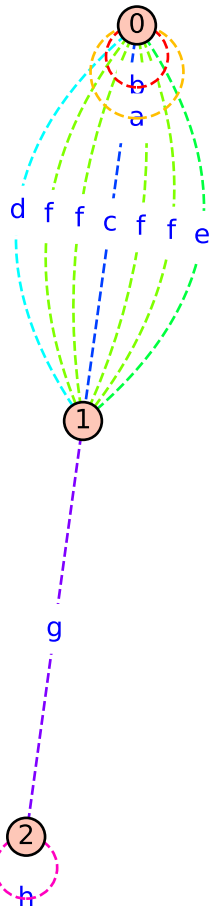

0
5
1
6
4
3
2


4
8
6
1
7
2
5
10
3
9
0


4
1
2
0
3


5
a
4
b
a
3
b
a
2
b
a
1
b
a
0
b
-1

```
sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
....:   (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: g.graphplot(edge_labels=True, color_by_label=True, edge_style='dashed').
↪plot()
Graphics object consisting of 26 graphics primitives
```



The `edge_style` option may be provided in the short format too:

```
sage: g.graphplot(edge_labels=True, color_by_label=True, edge_style='--').
↪plot()
Graphics object consisting of 26 graphics primitives
```

set_edges(***edge_options*)

Set the edge (or arrow) plotting parameters for the `GraphPlot` object.

This function is called by the constructor but can also be called to make updates to the vertex options of an existing `GraphPlot` object. Note that the changes are cumulative.

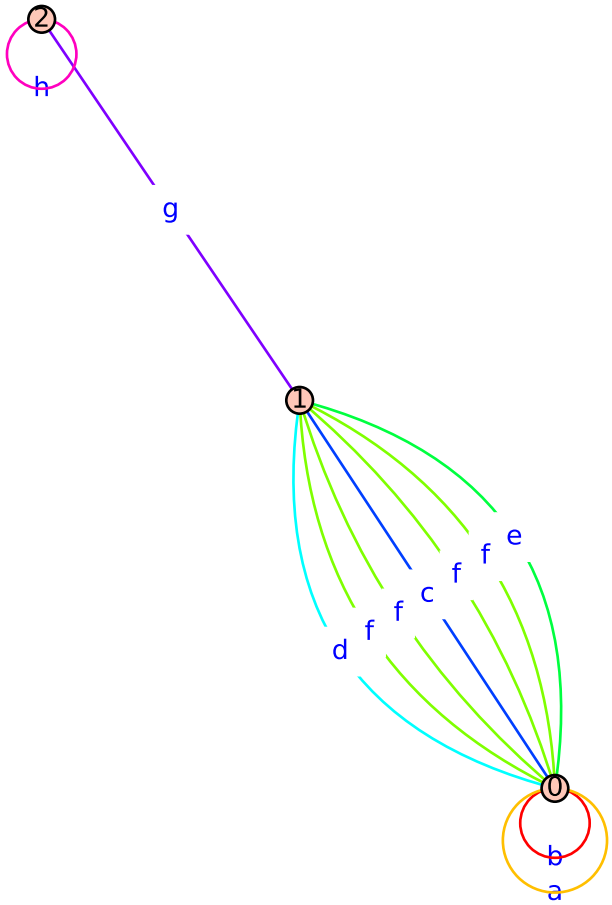
EXAMPLES:

```
sage: g = Graph(loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
....:   (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = g.graphplot(vertex_size=100, edge_labels=True, color_by_label=True,
```

(continues on next page)

(continued from previous page)

```
....:  edge_style='dashed')
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

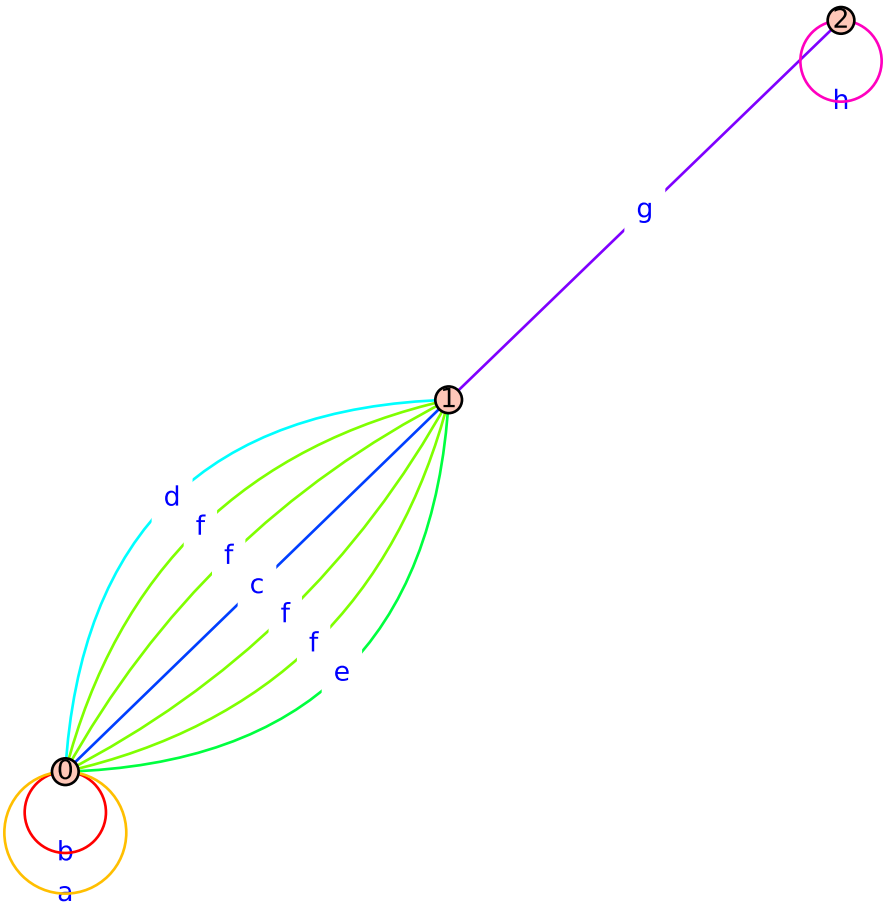


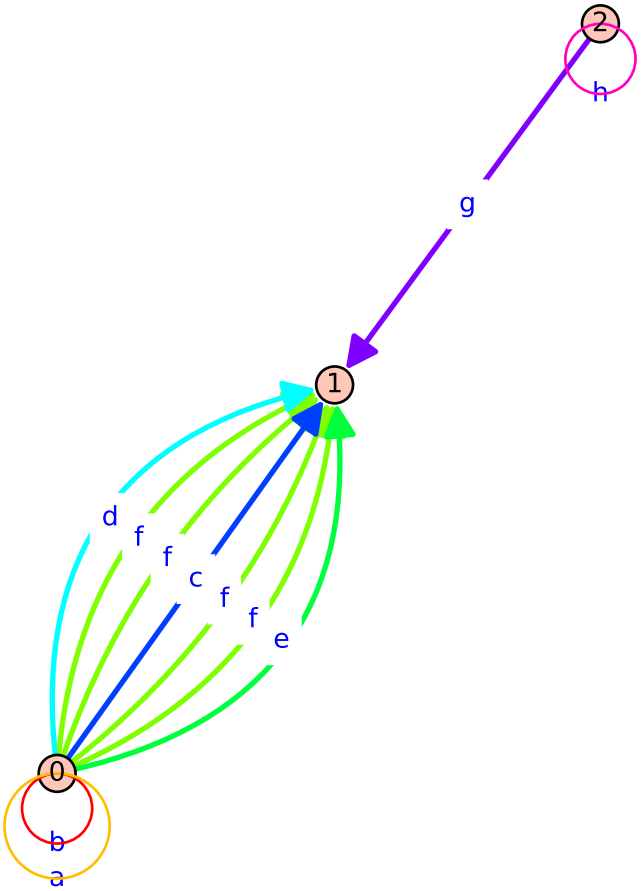

```
sage: GP.set_edges(edge_color='black')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

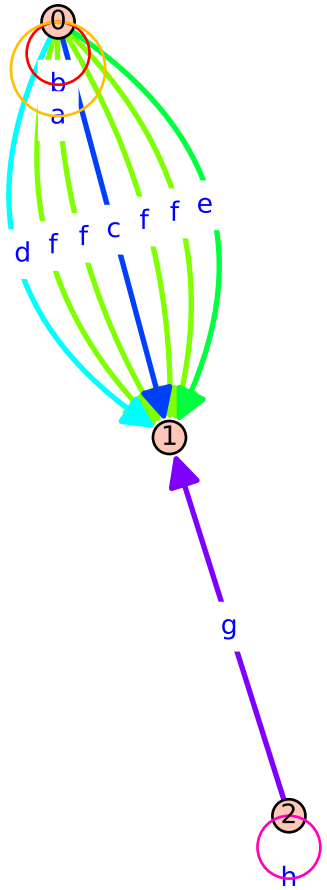
```
sage: d = DiGraph(loops=True, multiedges=True, sparse=True)
sage: d.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
....:   (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = d.graphplot(vertex_size=100, edge_labels=True, color_by_label=True,
....:  edge_style='dashed')
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
Graphics object consisting of 28 graphics primitives
```

```
sage: GP.set_edges(edge_color='black')
sage: GP.plot()
Graphics object consisting of 28 graphics primitives
```

set_pos()


2
h
g
1
d
f
f
c
f
f
e
0
b
a


2
h
g
1
d
f
f
c
f
f
e
0
b
a


0
b
a
d
f
f
c
f
f
e
1
g
2
h

Set the position plotting parameters for this GraphPlot.

EXAMPLES:

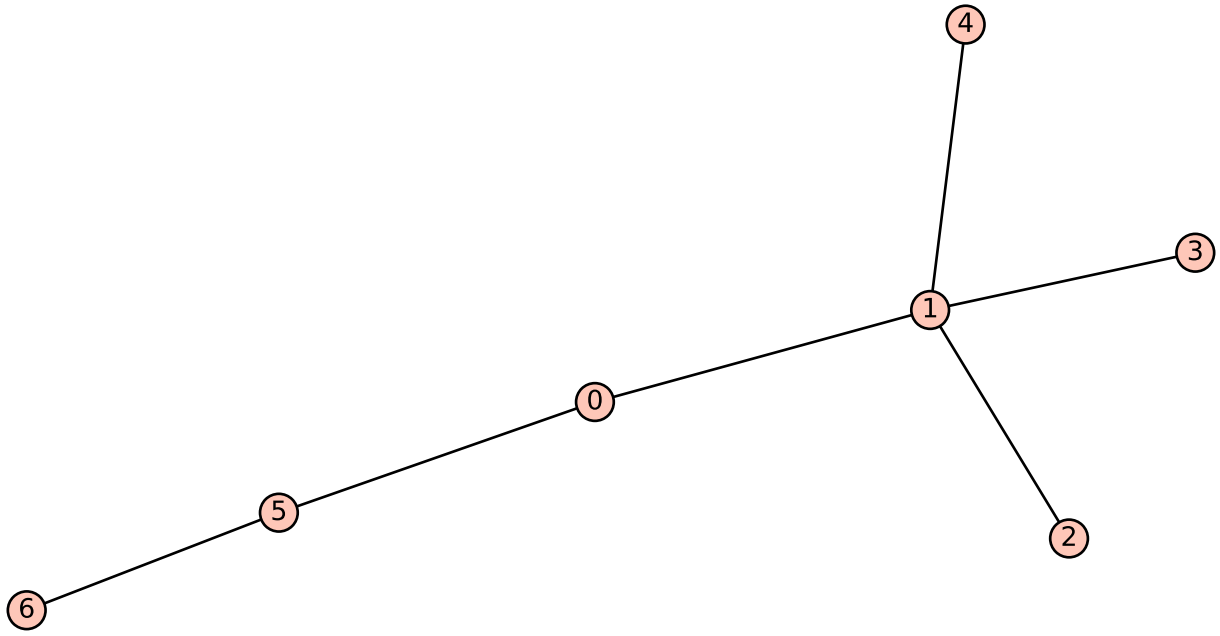
This function is called implicitly by the code below:

```
sage: g = Graph({0:[1,2], 2:[3], 4:[0,1]})
sage: g.graphplot(save_pos=True, layout='circular') # indirect doctest
GraphPlot object for Graph on 5 vertices
```

The following illustrates the format of a position dictionary, but due to numerical noise we do not check the values themselves:

```
sage: g.get_pos()
{0: (0.0, 1.0),
 1: (-0.951..., 0.309...),
 2: (-0.587..., -0.809...),
 3: (0.587..., -0.809...),
 4: (0.951..., 0.309...)}
```

```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]})
Graphics object consisting of 14 graphics primitives
```



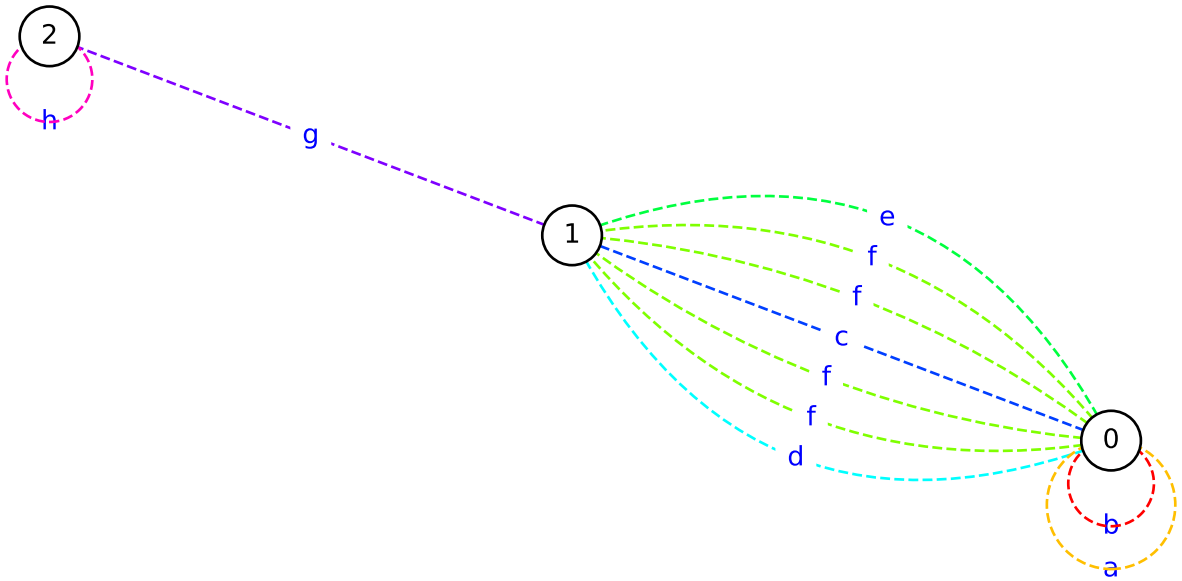
set_vertices(***vertex_options*)

Set the vertex plotting parameters for this `GraphPlot`.

This function is called by the constructor but can also be called to make updates to the vertex options of an existing `GraphPlot` object. Note that the changes are cumulative.

EXAMPLES:

```
sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
....:              (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = g.graphplot(vertex_size=100, edge_labels=True, color_by_label=True,
....:                  edge_style='dashed')
sage: GP.set_vertices(talk=True)
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
sage: GP.set_vertices(vertex_color='green', vertex_shape='^')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

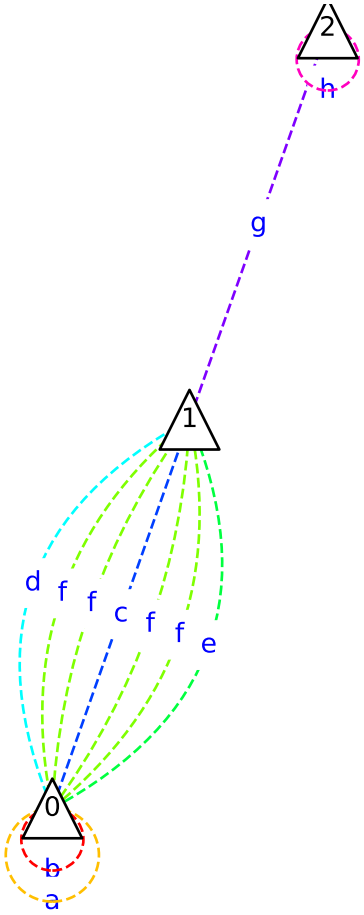


show(***kwds*)

Show the (Di)Graph associated with this `GraphPlot` object.

INPUT:

This method accepts all parameters of `sage.plot.graphics.Graphics.show()`.

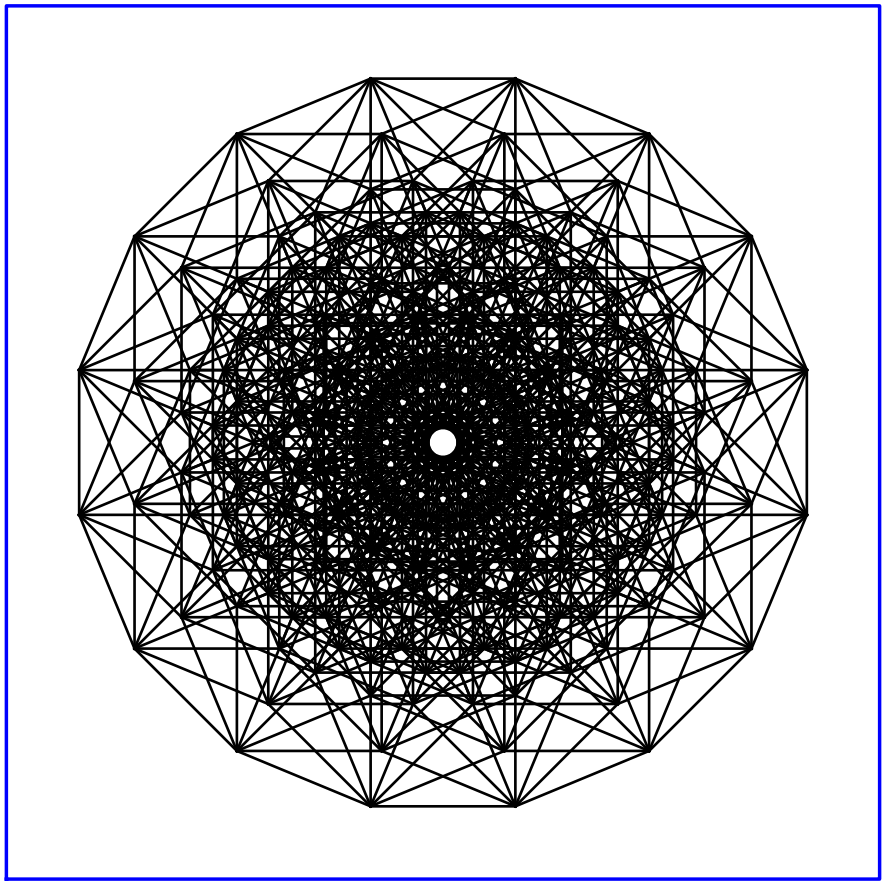

2
h
g
1
d
f
f
c
f
f
e
0
b
a

Note:

- See *the module's documentation* for information on default values of this method.
- Any options not used by plot will be passed on to the `show()` method.

EXAMPLES:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```



5.18 Graph plotting in Javascript with d3.js

This module implements everything that can be used to draw graphs with d3.js in Sage.

On Python's side, this is mainly done by wrapping a graph's edges and vertices in a structure that can then be used in the javascript code. This javascript code is then inserted into a .html file to be opened by a browser.

What Sage feeds javascript with is a "graph" object with the following content:

- `vertices` – each vertex is a dictionary defining :
  - `name` – The vertex's label

  - `group` – the vertex' color (integer)

The ID of a vertex is its index in the vertex list.

- `edges` – each edge is a dictionary defining :
  - `source` – the ID (int) of the edge's source
  - `target` – the ID (int) of the edge's destination
  - `color` – the edge's color (integer)
  - `value` – thickness of the edge
  - `strength` – the edge's strength in the automatic layout
  - `color` – color (hexadecimal code)
  - `curve` – distance from the barycenter of the two endpoints and the center of the edge. It defines the curve of the edge, which can be useful for multigraphs.
- `pos` – a list whose i th element is a dictionary defining the position of the i th vertex

It also contains the definition of some numerical/boolean variables whose definition can be found in the documentation of *`show()`* : `directed`, `charge`, `link_distance`, `link_strength`, `gravity`, `vertex_size`, `edge_thickness`.

Warning: Since the d3js package is not standard yet, the javascript is fetched from d3js.org website by the browser. If you want to avoid that (e.g. to protect your privacy or by lack of internet connection), you can install the d3js package for offline use by running `sage -i d3js` from the command line.

Todo:

- Add tooltip like in http://bl.ocks.org/bentwonk/2514276.
- Add a zoom through scrolling (http://bl.ocks.org/mbostock/3681006).

Authors:

- Nathann Cohen, Brice Onfroy – July 2013 – Initial version of the Sage code, Javascript code, using examples from d3.js.
- Thierry Monteil (June 2014): allow offline use of d3.js provided by d3js spkg.

5.18.1 Functions

`sage.graphs.graph_plot_js.`**`gen_html_code`**(*G*, *vertex_labels=True*, *edge_labels=False*, *vertex_partition=[]*, *vertex_colors=None*, *edge_partition=[]*, *force_spring_layout=False*, *charge=-120*, *link_distance=30*, *link_strength=2*, *gravity=0.04*, *vertex_size=7*, *edge_thickness=4*)

Creates a .html file showing the graph using d3.js.

This function returns the name of the .html file. If you want to visualize the actual graph use *`show()`*.

INPUT:

- `G` – the graph
- `vertex_labels` – boolean (default: `False`); whether to display vertex labels

- `edge_labels` – boolean (default: `False`); whether to display edge labels
- `vertex_partition` – list (default: `[]`); a list of lists representing a partition of the vertex set. Vertices are then colored in the graph according to the partition
- `vertex_colors` – dict (default: `None`); a dictionary representing a partition of the vertex set. Keys are colors (ignored) and values are lists of vertices. Vertices are then colored in the graph according to the partition
- `edge_partition` – list (default: `[]`); same as `vertex_partition`, with edges instead
- `force_spring_layout` – boolean (default: `False`); whether to take previously computed position of nodes into account if there is one, or to compute a spring layout
- `vertex_size` – integer (default: `7`); the size of a vertex' circle
- `edge_thickness` – integer (default: `4`); thickness of an edge
- `charge` – integer (default: `-120`); the vertices' charge. Defines how they repulse each other. See https://github.com/mbostock/d3/wiki/Force-Layout for more information
- `link_distance` – integer (default: `30`); see https://github.com/mbostock/d3/wiki/Force-Layout for more information
- `link_strength` – integer (default: `2`); see https://github.com/mbostock/d3/wiki/Force-Layout for more information
- `gravity` – float (default: `0.04`); see https://github.com/mbostock/d3/wiki/Force-Layout for more information

Warning: Since the d3js package is not standard yet, the javascript is fetched from d3js.org website by the browser. If you want to avoid that (e.g. to protect your privacy or by lack of internet connection), you can install the d3js package for offline use by running `sage -i d3js` from the command line.

EXAMPLES:

```
sage: graphs.RandomTree(50).show(method="js") # optional -- internet

sage: g = graphs.PetersenGraph()
sage: g.show(method="js", vertex_partition=g.coloring()) # optional -- internet

sage: graphs.DodecahedralGraph().show(method="js", force_spring_layout=True) #␣
↪optional -- internet

sage: graphs.DodecahedralGraph().show(method="js") # optional -- internet

sage: g = digraphs.DeBruijn(2, 2)
sage: g.allow_multiple_edges(True)
sage: g.add_edge("10", "10", "a")
sage: g.add_edge("10", "10", "b")
sage: g.add_edge("10", "10", "c")
sage: g.add_edge("10", "10", "d")
sage: g.add_edge("01", "11", "1")
sage: g.show(method="js", vertex_labels=True,edge_labels=True,
....:        link_distance=200, gravity=.05, charge=-500,
....:        edge_partition=[[("11", "12", "2"), ("21", "21", "a")]],
....:        edge_thickness=4) # optional -- internet
```

5.19 Vertex separation

This module implements several algorithms to compute the vertex separation of a digraph and the corresponding ordering of the vertices. It also implements tests functions for evaluation the width of a linear ordering.

Given an ordering $v_1, \cdots, v_n$ of the vertices of $V(G)$, its *cost* is defined as:

$$c(v_1, ..., v_n) = \max_{1 \leq i \leq n} c'(\{v_1, ..., v_i\})$$

Where

$$c'(S) = |N_G^+(S) \backslash S|$$

The *vertex separation* of a digraph G is equal to the minimum cost of an ordering of its vertices.

Vertex separation and pathwidth

The vertex separation is defined on a digraph, but one can obtain from a graph G a digraph D with the same vertex set, and in which each edge uv of G is replaced by two edges uv and vu in D. The vertex separation of D is equal to the pathwidth of G, and the corresponding ordering of the vertices of D, also called a *layout*, encodes an optimal path-decomposition of G. This is a result of Kinnersley [Kin1992] and Bodlaender [Bod1998].

This module contains the following methods

pathwidth()	Compute the pathwidth of `self` (and provides a decomposition)
path_decomposition()	Return the pathwidth of the given graph and the ordering of the vertices resulting in a corresponding path decomposition
vertex_separation()	Return an optimal ordering of the vertices and its cost for vertex-separation
vertex_separation_exp()	Compute the vertex separation of G using an exponential time and space algorithm
vertex_separation_MILP()	Compute the vertex separation of G and the optimal ordering of its vertices using an MILP formulation
vertex_separation_BAB()	Compute the vertex separation of G and the optimal ordering of its vertices using a branch and bound algorithm
lower_bound()	Return a lower bound on the vertex separation of G
is_valid_ordering()	Test if the linear vertex ordering L is valid for (di)graph G
width_of_path_decomposition()	Return the width of the path decomposition induced by the linear ordering L of the vertices of G
linear_ordering_to_path_decomposition()	Return the path decomposition encoded in the ordering L

5.19.1 Exponential algorithm for vertex separation

In order to find an optimal ordering of the vertices for the vertex separation, this algorithm tries to save time by computing the function $c'(S)$ **at most once** once for each of the sets $S \subseteq V(G)$. These values are stored in an array of size 2^n where reading the value of $c'(S)$ or updating it can be done in constant (and small) time.

Assuming that we can compute the cost of a set S and remember it, finding an optimal ordering is an easy task. Indeed, we can think of the sequence $v_1, ..., v_n$ of vertices as a sequence of *sets* $\{v_1\}, \{v_1, v_2\}, ..., \{v_1, ..., v_n\}$, whose cost is precisely $\max c'(\{v_1\}), c'(\{v_1, v_2\}), ..., c'(\{v_1, ..., v_n\})$. Hence, when considering the digraph on the 2^n sets $S \subseteq V(G)$ where there is an arc from S to S' if $S' = S \cap \{v\}$ for some v (that is, if the sets S and S' can be consecutive in a sequence), an ordering of the vertices of G corresponds to a *path* from $\emptyset$ to $\{v_1, ..., v_n\}$. In this setting, checking whether there exists a ordering of cost less than k can be achieved by checking whether there exists a directed path $\emptyset$ to $\{v_1, ..., v_n\}$ using only sets of cost less than k. This is just a depth-first-search, for each k.

Lazy evaluation of c'

In the previous algorithm, most of the time is actually spent on the computation of $c'(S)$ for each set $S \subseteq V(G)$ – i.e. 2^n computations of neighborhoods. This can be seen as a huge waste of time when noticing that it is useless to know that the value $c'(S)$ for a set S is less than k if all the paths leading to S have a cost greater than k. For this reason, the value of $c'(S)$ is computed lazily during the depth-first search. Explanation :

When the depth-first search discovers a set of size less than k, the costs of its out-neighbors (the potential sets that could follow it in the optimal ordering) are evaluated. When an out-neighbor is found that has a cost smaller than k, the depth-first search continues with this set, which is explored with the hope that it could lead to a path toward $\{v_1, ..., v_n\}$. On the other hand, if an out-neighbour has a cost larger than k it is useless to attempt to build a cheap sequence going though this set, and the exploration stops there. This way, a large number of sets will never be evaluated and *a lot* of computational time is saved this way.

Besides, some improvement is also made by "improving" the values found by c'. Indeed, $c'(S)$ is a lower bound on the cost of a sequence containing the set S, but if all out-neighbors of S have a cost of $c'(S) + 5$ then one knows that having S in a sequence means a total cost of at least $c'(S) + 5$. For this reason, for each set S we store the value of $c'(S)$, and replace it by $\max(c'(S), \min_{\text{next}})$ (where $\min_{\text{next}}$ is the minimum of the costs of the out-neighbors of S) once the costs of these out-neighbors have been evaluated by the algorithm.

Note: Because of its current implementation, this algorithm only works on graphs on less than 32 vertices. This can be changed to 64 if necessary, but 32 vertices already require 4GB of memory. Running it on 64 bits is not expected to be doable by the computers of the next decade : $-D$

Lower bound on the vertex separation

One can obtain a lower bound on the vertex separation of a graph in exponential time but *small* memory by computing once the cost of each set S. Indeed, the cost of a sequence $v_1, ..., v_n$ corresponding to sets $\{v_1\}, \{v_1, v_2\}, ..., \{v_1, ..., v_n\}$ is

$$\max c'(\{v_1\}), c'(\{v_1, v_2\}), ..., c'(\{v_1, ..., v_n\}) \geq \max c'_1, ..., c'_n$$

where c_i is the minimum cost of a set S on i vertices. Evaluating the c_i can take time (and in particular more than the previous exact algorithm), but it does not need much memory to run.

5.19.2 MILP formulation for the vertex separation

We describe below a mixed integer linear program (MILP) for determining an optimal layout for the vertex separation of G, which is an improved version of the formulation proposed in [SP2010]. It aims at building a sequence S_t of sets such that an ordering $v_1, ..., v_n$ of the vertices correspond to $S_0 = \{v_1\}, S_2 = \{v_1, v_2\}, ..., S_{n-1} = \{v_1, ..., v_n\}$.

Variables:

- y_v^t – Variable set to 1 if $v \in S_t$, and 0 otherwise. The order of v in the layout is the smallest t such that $y_v^t == 1$.
- u_v^t – Variable set to 1 if $v \notin S_t$ and v has an in-neighbor in S_t. It is set to 0 otherwise.
- x_v^t – Variable set to 1 if either $v \in S_t$ or if v has an in-neighbor in S_t. It is set to 0 otherwise.
- z – Objective value to minimize. It is equal to the maximum over all step t of the number of vertices such that $u_v^t == 1$.

MILP formulation:

$$\begin{aligned}
\text{Minimize:} & z && && (5.1)\\
\text{Such that:} & x_v^t \leq x_v^{t+1} && \forall v \in V,\ 0 \leq t \leq n-2 && (5.2)\\
& y_v^t \leq y_v^{t+1} && \forall v \in V,\ 0 \leq t \leq n-2 && (5.3)\\
& y_v^t \leq x_w^t && \forall v \in V,\ \forall w \in N^+(v),\ 0 \leq t \leq n-1 && (5.4)\\
& \sum_{v \in V} y_v^t = t+1 && 0 \leq t \leq n-1 && (5.5)\\
& x_v^t - y_v^t \leq u_v^t && \forall v \in V,\ 0 \leq t \leq n-1 && (5.6)\\
& \sum_{v \in V} u_v^t \leq z && 0 \leq t \leq n-1 && (5.7)\\
& 0 \leq x_v^t \leq 1 && \forall v \in V,\ 0 \leq t \leq n-1 && (5.8)\\
& 0 \leq u_v^t \leq 1 && \forall v \in V,\ 0 \leq t \leq n-1 && (5.9)\\
& y_v^t \in \{0,1\} && \forall v \in V,\ 0 \leq t \leq n-1 && (5.10)\\
& 0 \leq z \leq n && && (5.11)
\end{aligned}$$

The vertex separation of G is given by the value of z, and the order of vertex v in the optimal layout is given by the smallest t for which $y_v^t == 1$.

5.19.3 Branch and Bound algorithm for the vertex separation

We describe below the principle of a branch and bound algorithm (BAB) for determining an optimal ordering for the vertex separation of G, as proposed in [CMN2014].

Greedy steps:

Let us denote $\mathcal{L}(S)$ the set of all possible orderings of the vertices in S, and let $\mathcal{L}_P(S) \subseteq \mathcal{L}(S)$ be the orderings starting with a prefix P. Let also $c(L)$ be the cost of the ordering $L \in \mathcal{L}(V)$ as defined above.

Given a digraph $D = (V, A)$, a set $S \subset V$, and a prefix P, it has been proved in [CMN2014] that $\min_{L \in \mathcal{L}_P(V)} c(L) = \min_{L \in \mathcal{L}_{P+v}(V)} c(L)$ holds in two (non exhaustive) cases:

$$\text{or } \begin{cases} N^+(v) \subseteq S \cup N^+(S) \\ v \in N^+(S) \text{ and } N^+(v) \setminus (S \cup N^+(S)) = \{w\} \end{cases}$$

In other words, if we find a vertex v satisfying the above conditions, the best possible ordering with prefix P has the same cost as the best possible ordering with prefix $P+v$. So we can greedily extend the prefix with vertices satisfying the conditions which results in a significant reduction of the search space.

The algorithm:

Given the current prefix P and the current upper bound UB (either an input upper bound or the cost of the best solution found so far), apply the following steps:

- Extend the prefix P into a prefix P' using the greedy steps as described above.
- Sort the vertices $v \in V \setminus P'$ by increasing values of $|N^+(P+v)|$, and prune the vertices with a value larger or equal to UB. Let Δ be the resulting sorted list.
- Repeat with prefix $P' + v$ for all $v \in \Delta$ and keep the best found solution.

If a lower bound is passed to the algorithm, it will stop as soon as a solution with cost equal to that lower bound is found.

Storing prefixes:

If for a prefix P we have $c(P) < \min_{L \in \mathcal{L}_P(V)} c(L) = C$, then for any permutation P' of P we have $\min_{L \in \mathcal{L}_{P'}(V)} c(L) \geq C$.

Thus, given such a prefix P there is no need to explore any of the orderings starting with one of its permutations. To do so, we store P (as a set of vertices) to cut branches later. See [CMN2014] for more details.

Since the number of stored sets can get very large, one can control the maximum length and the maximum number of stored prefixes.

5.19.4 Authors

- Nathann Cohen (2011-10): Initial version and exact exponential algorithm
- David Coudert (2012-04): MILP formulation and tests functions
- David Coudert (2015-01): BAB formulation and tests functions

5.19.5 Methods

sage.graphs.graph_decompositions.vertex_separation.**is_valid_ordering**(G, L)

Test if the linear vertex ordering L is valid for (di)graph G.

A linear ordering L of the vertices of a (di)graph G is valid if all vertices of G are in L, and if L contains no other vertex and no duplicated vertices.

INPUT:

- `G` – a Graph or a DiGraph.
- `L` – an ordered list of the vertices of `G`.

OUTPUT:

Returns `True` if L is a valid vertex ordering for G, and `False` otherwise.

EXAMPLES:

Path decomposition of a cycle:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = graphs.CycleGraph(6)
sage: L = [u for u in G.vertices()]
sage: vertex_separation.is_valid_ordering(G, L)
True
sage: vertex_separation.is_valid_ordering(G, [1,2])
False
```

sage.graphs.graph_decompositions.vertex_separation.**linear_ordering_to_path_decomposition**(G, L)

Return the path decomposition encoded in the ordering L

INPUT:

- `G` – a Graph
- `L` – a linear ordering for G

OUTPUT:

A path graph whose vertices are the bags of the path decomposition.

EXAMPLES:

The bags of an optimal path decomposition of a path-graph have two vertices each:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_
↪separation
sage: from sage.graphs.graph_decompositions.vertex_separation import linear_
↪ordering_to_path_decomposition
sage: g = graphs.PathGraph(5)
sage: pw, L = vertex_separation(g, algorithm = "BAB"); pw
1
sage: h = linear_ordering_to_path_decomposition(g, L)
sage: sorted(h, key=str)
[{0, 1}, {1, 2}, {2, 3}, {3, 4}]
sage: sorted(h.edge_iterator(labels=None), key=str)
[({0, 1}, {1, 2}), ({1, 2}, {2, 3}), ({2, 3}, {3, 4})]
```

Giving a non-optimal linear ordering:

```
sage: g = graphs.PathGraph(5)
sage: L = [1, 4, 0, 2, 3]
sage: from sage.graphs.graph_decompositions.vertex_separation import width_of_
↪path_decomposition
sage: width_of_path_decomposition(g, L)
3
sage: h = linear_ordering_to_path_decomposition(g, L)
sage: h.vertices()
[{0, 2, 3, 4}, {0, 1, 2}]
```

The bags of the path decomposition of a cycle have three vertices each:

```
sage: g = graphs.CycleGraph(6)
sage: pw, L = vertex_separation(g, algorithm = "BAB"); pw
2
sage: h = linear_ordering_to_path_decomposition(g, L)
sage: sorted(h, key=str)
[{0, 1, 5}, {1, 2, 5}, {2, 3, 4}, {2, 4, 5}]
sage: sorted(h.edge_iterator(labels=None), key=str)
[({0, 1, 5}, {1, 2, 5}), ({1, 2, 5}, {2, 4, 5}), ({2, 4, 5}, {2, 3, 4})]
```

sage.graphs.graph_decompositions.vertex_separation.**lower_bound**(G)

Return a lower bound on the vertex separation of G.

INPUT:

- `G` – a Graph or a DiGraph

OUTPUT:

A lower bound on the vertex separation of D (see the module's documentation).

Note: This method runs in exponential time but has no memory constraint.

EXAMPLES:

On a circuit:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import lower_bound
sage: g = digraphs.Circuit(6)
sage: lower_bound(g)
1
```

sage.graphs.graph_decompositions.vertex_separation.**path_decomposition**(*G*, *algorithm='BAB'*, *cut_off=None*, *upper_bound=None*, *verbose=False*, *max_prefix_length=20*, *max_prefix_number=1000000*)

Return the pathwidth of the given graph and the ordering of the vertices resulting in a corresponding path decomposition.

INPUT:

- `G` – a Graph
- `algorithm` – string (default: `"BAB"`); algorithm to use among:
  - `"BAB"` – Use a branch-and-bound algorithm. This algorithm has no size restriction but could take a very long time on large graphs. It can also be used to test is the input (di)graph has vertex separation at most `upper_bound` or to return the first found solution with vertex separation less or equal to a `cut_off` value.
  - `exponential` – Use an exponential time and space algorithm. This algorithm only works of graphs on less than 32 vertices.
  - `MILP` – Use a mixed integer linear programming formulation. This algorithm has no size restriction but could take a very long time.
- `upper_bound` – integer (default: `None`); parameter used by the `"BAB"` algorithm. If specified, the algorithm searches for a solution with `width < upper_bound`. It helps cutting branches. However, if the given upper bound is too low, the algorithm may not be able to find a solution.
- `cut_off` – integer (default: `None`); parameter used by the `"BAB"` algorithm. This bound allows us to stop the search as soon as a solution with width at most `cut_off` is found, if any. If this bound cannot be reached, the best solution found is returned, unless a too low `upper_bound` is given.
- `verbose` – boolean (default: `False`); whether to display information on the computations
- `max_prefix_length` – integer (default: 20); limits the length of the stored prefixes to prevent storing too many prefixes. This parameter is used only when `algorithm=="BAB"`.
- `max_prefix_number` – integer (default: 10**6); upper bound on the number of stored prefixes used to prevent using too much memory. This parameter is used only when `algorithm=="BAB"`.

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

See also:

- *`Graph.treewidth()`* – computes the treewidth of a graph

EXAMPLES:

The pathwidth of a cycle is equal to 2:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import path_
↪decomposition
sage: g = graphs.CycleGraph(6)
```

(continues on next page)

(continued from previous page)

```
sage: pw, L = path_decomposition(g, algorithm = "BAB"); pw
2
sage: pw, L = path_decomposition(g, algorithm = "exponential"); pw
2
sage: pw, L = path_decomposition(g, algorithm = "MILP"); pw
2
```

sage.graphs.graph_decompositions.vertex_separation.**pathwidth**(*self*, *k=None*, *certificate=False*, *algorithm='BAB'*, *verbose=False*, *max_prefix_length=20*, *max_prefix_number=1000000*)

Compute the pathwidth of `self` (and provides a decomposition)

INPUT:

- `k` – integer (default: `None`); the width to be considered. When `k` is an integer, the method checks that the graph has pathwidth $\leq k$. If `k` is `None` (default), the method computes the optimal pathwidth.
- `certificate` – boolean (default: `False`); whether to return the path-decomposition itself
- `algorithm` – string (default: `"BAB"`); algorithm to use among:
  - `"BAB"` – Use a branch-and-bound algorithm. This algorithm has no size restriction but could take a very long time on large graphs. It can also be used to test is the input graph has pathwidth $\leq k$, in which cas it will return the first found solution with width $\leq k$ is `certificate==True`.
  - `exponential` – Use an exponential time and space algorithm. This algorithm only works of graphs on less than 32 vertices.
  - `MILP` – Use a mixed integer linear programming formulation. This algorithm has no size restriction but could take a very long time.
- `verbose` – boolean (default: `False`); whether to display information on the computations
- `max_prefix_length` – integer (default: 20); limits the length of the stored prefixes to prevent storing too many prefixes. This parameter is used only when `algorithm=="BAB"`.
- `max_prefix_number` – integer (default: 10**6); upper bound on the number of stored prefixes used to prevent using too much memory. This parameter is used only when `algorithm=="BAB"`.

OUTPUT:

Return the pathwidth of `self`. When `k` is specified, it returns `False` when no path-decomposition of width $\leq k$ exists or `True` otherwise. When `certificate=True`, the path-decomposition is also returned.

See also:

- *`Graph.treewidth()`* – computes the treewidth of a graph
- *`vertex_separation()`* – computes the vertex separation of a (di)graph

EXAMPLES:

The pathwidth of a cycle is equal to 2:

```
sage: g = graphs.CycleGraph(6)
sage: g.pathwidth()
2
sage: pw, decomp = g.pathwidth(certificate=True)
```

(continues on next page)

(continued from previous page)

```
sage: sorted(decomp, key=str)
[{0, 1, 5}, {1, 2, 5}, {2, 3, 4}, {2, 4, 5}]
```

The pathwidth of a Petersen graph is 5:

```
sage: g = graphs.PetersenGraph()
sage: g.pathwidth()
5
sage: g.pathwidth(k=2)
False
sage: g.pathwidth(k=6)
True
sage: g.pathwidth(k=6, certificate=True)
(True, Graph on 5 vertices)
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation**(*G*, *algorithm='BAB'*, *cut_off=None*, *upper_bound=None*, *verbose=False*, *max_prefix_length=20*, *max_prefix_number=1000000*)

Return an optimal ordering of the vertices and its cost for vertex-separation.

INPUT:

- `G` – a Graph or a DiGraph
- `algorithm` – string (default: `"BAB"`); algorithm to use among:
  - `"BAB"` – Use a branch-and-bound algorithm. This algorithm has no size restriction but could take a very long time on large graphs. It can also be used to test is the input (di)graph has vertex separation at most `upper_bound` or to return the first found solution with vertex separation less or equal to a `cut_off` value.
  - `exponential` – Use an exponential time and space algorithm. This algorithm only works of graphs on less than 32 vertices.
  - `MILP` – Use a mixed integer linear programming formulation. This algorithm has no size restriction but could take a very long time.
- `upper_bound` – integer (default: `None`); parameter used by the `"BAB"` algorithm. If specified, the algorithm searches for a solution with `width < upper_bound`. It helps cutting branches. However, if the given upper bound is too low, the algorithm may not be able to find a solution.
- `cut_off` – integer (default: `None`); parameter used by the `"BAB"` algorithm. This bound allows us to stop the search as soon as a solution with width at most `cut_off` is found, if any. If this bound cannot be reached, the best solution found is returned, unless a too low `upper_bound` is given.
- `verbose` – boolean (default: `False`); whether to display information on the computations
- `max_prefix_length` – integer (default: 20); limits the length of the stored prefixes to prevent storing too many prefixes. This parameter is used only when `algorithm=="BAB"`.
- `max_prefix_number` – integer (default: 10**6); upper bound on the number of stored prefixes used to prevent using too much memory. This parameter is used only when `algorithm=="BAB"`.

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

EXAMPLES:

Comparison of methods:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_
↪separation
sage: G = digraphs.DeBruijn(2,3)
sage: vs,L = vertex_separation(G, algorithm="BAB"); vs
2
sage: vs,L = vertex_separation(G, algorithm="exponential"); vs
2
sage: vs,L = vertex_separation(G, algorithm="MILP"); vs
2
sage: G = graphs.Grid2dGraph(3,3)
sage: vs,L = vertex_separation(G, algorithm="BAB"); vs
3
sage: vs,L = vertex_separation(G, algorithm="exponential"); vs
3
sage: vs,L = vertex_separation(G, algorithm="MILP"); vs
3
```

Digraphs with multiple strongly connected components:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_
↪separation
sage: D = digraphs.Path(8)
sage: print(vertex_separation(D))
(0, [7, 6, 5, 4, 3, 2, 1, 0])
sage: D = DiGraph( random_DAG(30) )
sage: vs,L = vertex_separation(D); vs
0
sage: K4 = DiGraph( graphs.CompleteGraph(4) )
sage: D = K4+K4
sage: D.add_edge(0, 4)
sage: print(vertex_separation(D))
(3, [4, 5, 6, 7, 0, 1, 2, 3])
sage: D = K4+K4+K4
sage: D.add_edge(0, 4)
sage: D.add_edge(0, 8)
sage: print(vertex_separation(D))
(3, [8, 9, 10, 11, 4, 5, 6, 7, 0, 1, 2, 3])
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation_BAB**(*G*, *cut_off=None*, *upper_bound=None*, *max_prefix_length=20*, *max_prefix_number=1000*, *verbose=False*)

Branch and Bound algorithm for the vertex separation.

This method implements the branch and bound algorithm for the vertex separation of directed graphs and the pathwidth of undirected graphs proposed in [CMN2014]. The implementation is valid for both Graph and DiGraph. See the documentation of the *vertex_separation* module.

INPUT:

- `G` – a Graph or a DiGraph.
- `cut_off` – integer (default: `None`); bound to consider in the branch and bound algorithm. This allows us to stop the search as soon as a solution with width at most `cut_off` is found, if any. If this bound cannot be reached, the best solution found is returned, unless a too low `upper_bound` is given.
- `upper_bound` – integer (default: `None`); if specified, the algorithm searches for a solution with `width < upper_bound`. It helps cutting branches. However, if the given upper bound is too low, the algorithm may not be able to find a solution.
- `max_prefix_length` – integer (default: 20); limits the length of the stored prefixes to prevent storing too many prefixes
- `max_prefix_number` – integer (default: 10**6); upper bound on the number of stored prefixes used to prevent using too much memory
- `verbose` – boolean (default: `False`); display some information when set to `True`

OUTPUT:

- `width` – the computed vertex separation
- `seq` – an ordering of the vertices of width `width`

EXAMPLES:

The algorithm is valid for the vertex separation:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: D = digraphs.RandomDirectedGNP(15, .2)
sage: vb, seqb = VS.vertex_separation_BAB(D)
sage: vd, seqd = VS.vertex_separation_exp(D)
sage: vb == vd
True
sage: vb == VS.width_of_path_decomposition(D, seqb)
True
```

The vertex separation of a $N \times N$ grid is N:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.Grid2dGraph(4,4)
sage: vs, seq = VS.vertex_separation_BAB(G); vs
4
sage: vs == VS.width_of_path_decomposition(G, seq)
True
```

The vertex separation of a $N \times M$ grid with $N < M$ is N:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.Grid2dGraph(3,5)
sage: vs, seq = VS.vertex_separation_BAB(G); vs
3
sage: vs == VS.width_of_path_decomposition(G, seq)
True
```

The vertex separation of circuit of order $N \geq 2$ is 1:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: D = digraphs.Circuit(10)
```

(continues on next page)

(continued from previous page)

```
sage: vs, seq = VS.vertex_separation_BAB(D); vs
1
sage: vs == VS.width_of_path_decomposition(D, seq)
True
```

The vertex separation of cycle of order $N \geq 3$ is 2:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.CycleGraph(10)
sage: vs, seq = VS.vertex_separation_BAB(G); vs
2
```

The vertex separation of `MycielskiGraph(5)` is 10:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.MycielskiGraph(5)
sage: vs, seq = VS.vertex_separation_BAB(G); vs
10
```

Searching for any solution with width less or equal to `cut_off`:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.MycielskiGraph(5)
sage: VS.vertex_separation_BAB(G, cut_off=11)[0] <= 11
True
sage: VS.vertex_separation_BAB(G, cut_off=10)[0] <= 10
True
sage: VS.vertex_separation_BAB(G, cut_off=9)[0] <= 9
False
```

Testing for the existence of a solution with width strictly less than `upper_bound`:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.MycielskiGraph(5)
sage: vs, seq = VS.vertex_separation_BAB(G, upper_bound=11); vs
10
sage: vs, seq = VS.vertex_separation_BAB(G, upper_bound=10); vs
-1
sage: vs, seq = VS.vertex_separation_BAB(G, cut_off=11, upper_bound=10); vs
-1
```

Changing the parameters of the prefix storage:

```
sage: from sage.graphs.graph_decompositions import vertex_separation as VS
sage: G = graphs.MycielskiGraph(5)
sage: vs, seq = VS.vertex_separation_BAB(G, max_prefix_length=0); vs
10
sage: vs, seq = VS.vertex_separation_BAB(G, max_prefix_number=5); vs
10
sage: vs, seq = VS.vertex_separation_BAB(G, max_prefix_number=0); vs
10
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation_MILP**(*G*, *integrality=False*, *solver=None*, *verbosity=0*)

Compute the vertex separation of G and the optimal ordering of its vertices using an MILP formulation.

This function uses a mixed integer linear program (MILP) for determining an optimal layout for the vertex separation of G. This MILP is an improved version of the formulation proposed in [SP2010]. See the *module's documentation* for more details on this MILP formulation.

INPUT:

- `G` – a Graph or a DiGraph
- `integrality` – boolean (default: `False`); specify if variables x_v^t and u_v^t must be integral or if they can be relaxed. This has no impact on the validity of the solution, but it is sometimes faster to solve the problem using binary variables only.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbosity` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

EXAMPLES:

Vertex separation of a De Bruijn digraph:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = digraphs.DeBruijn(2,3)
sage: vs, L = vertex_separation.vertex_separation_MILP(G); vs
2
sage: vs == vertex_separation.width_of_path_decomposition(G, L)
True
sage: vse, Le = vertex_separation.vertex_separation(G); vse
2
```

The vertex separation of a circuit is 1:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = digraphs.Circuit(6)
sage: vs, L = vertex_separation.vertex_separation_MILP(G); vs
1
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation_exp**(*G*, *verbose=False*)

Return an optimal ordering of the vertices and its cost for vertex-separation.

INPUT:

- `G` – a Graph or a DiGraph
- `verbose` – boolean (default: `False`); whether to display information on the computations

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

Note: Because of its current implementation, this algorithm only works on graphs on less than 32 vertices. This can be changed to 54 if necessary, but 32 vertices already require 4GB of memory.

EXAMPLES:

The vertex separation of a circuit is equal to 1:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_
↪separation_exp
sage: g = digraphs.Circuit(6)
sage: vertex_separation_exp(g)
(1, [0, 1, 2, 3, 4, 5])
```

sage.graphs.graph_decompositions.vertex_separation.**width_of_path_decomposition**(*G*, *L*)

Return the width of the path decomposition induced by the linear ordering L of the vertices of G.

If G is an instance of *Graph*, this function returns the width $pw_L(G)$ of the path decomposition induced by the linear ordering L of the vertices of G. If G is a *DiGraph*, it returns instead the width $vs_L(G)$ of the directed path decomposition induced by the linear ordering L of the vertices of G, where

$$vs_L(G) = \max_{0 \leq i < |V|-1} |N^+(L[:i]) \setminus L[:i]|$$
$$pw_L(G) = \max_{0 \leq i < |V|-1} |N(L[:i]) \setminus L[:i]|$$

INPUT:

- `G` – a Graph or a DiGraph
- `L` – a linear ordering of the vertices of `G`

EXAMPLES:

Path decomposition of a cycle:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = graphs.CycleGraph(6)
sage: L = [u for u in G.vertices()]
sage: vertex_separation.width_of_path_decomposition(G, L)
2
```

Directed path decomposition of a circuit:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = digraphs.Circuit(6)
sage: L = [u for u in G.vertices()]
sage: vertex_separation.width_of_path_decomposition(G, L)
1
```

5.20 Rank Decompositions of graphs

This modules wraps a C code from Philipp Klaus Krause computing a an optimal rank-decomposition.

Definitions :

Given a graph G and a subset $S \subseteq V(G)$ of vertices, the *rank-width* of S in G, denoted $rw_G(S)$, is equal to the rank in $GF(2)$ of the $|S| \times (|V| - |S|)$ matrix of the adjacencies between the vertices of S and $V \backslash S$. By definition, $rw_G(S)$ is equal to $rw_G(\overline{S})$ where $\overline{S}$ is the complement of S in $V(G)$.

A *rank-decomposition* of G is a tree whose n leaves are the elements of $V(G)$, and whose internal nodes have degree 3. In a tree, any edge naturally corresponds to a bipartition of the vertex set : indeed, the removal of any edge splits the tree into two connected components, thus splitting the set of leaves (i.e. vertices of G) into two sets. Hence we can define for any edge $e \in E(G)$ a width equal to the value $rw_G(S)$ or $rw_G(\overline{S})$, where $S, \overline{S}$ is the bipartition obtained from e. The *rank-width* associated to the whole decomposition is then set to the maximum of the width of all the edges it contains.

A *rank-decomposition* is said to be optimal for G if it is the decomposition achieving the minimal *rank-width*.

RW – The original source code :

RW is a program that calculates rank-width and rank-decompositions. It is based on ideas from :

- "Computing rank-width exactly" by Sang-il Oum [Oum2009]
- "Sopra una formula numerica" by Ernesto Pascal
- "Generation of a Vector from the Lexicographical Index" by B.P. Buckles and M. Lybanon [BL1977]
- "Fast additions on masked integers" by Michael D. Adams and David S. Wise [AW2006]

OUTPUT:

The rank decomposition is returned as a tree whose vertices are subsets of $V(G)$. Its leaves, corresponding to the vertices of G are sets of 1 elements, i.e. singletons.

```
sage: g = graphs.PetersenGraph()
sage: rw, tree = g.rank_decomposition()
sage: all(len(v)==1 for v in tree if tree.degree(v) == 1)
True
```

The internal nodes are sets of the decomposition. This way, it is easy to deduce the bipartition associated to an edge from the tree. Indeed, two adjacent vertices of the tree are comparable sets : they yield the bipartition obtained from the smaller of the two and its complement.

```
sage: g = graphs.PetersenGraph()
sage: rw, tree = g.rank_decomposition()
sage: u = Set([8, 9, 3, 7])
sage: v = Set([8, 9])
sage: tree.has_edge(u,v)
True
sage: m = min(u,v)
sage: bipartition = (m, Set(g.vertices()) - m)
sage: bipartition
({8, 9}, {0, 1, 2, 3, 4, 5, 6, 7})
```

Warning:

- The current implementation cannot handle graphs of ≥ 32 vertices.
- A bug that has been reported upstream make the code crash immediately on instances of size 30. If you experience this kind of bug please report it to us, what we need is some information on the hardware you run to know where it comes from !

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.rank_decomposition()
(3, Graph on 19 vertices)
```

AUTHORS:

- Philipp Klaus Krause : Implementation of the C algorithm
- Nathann Cohen : Interface with Sage and documentation

5.20.1 Methods

sage.graphs.graph_decompositions.rankwidth.**mkgraph**(*num_vertices*)

Return the graph corresponding to the current rank-decomposition.

(This function is for internal use)

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.rankwidth import rank_decomposition
sage: g = graphs.PetersenGraph()
sage: rank_decomposition(g)
(3, Graph on 19 vertices)
```

sage.graphs.graph_decompositions.rankwidth.**rank_decomposition**(*G*, *verbose=False*)

Compute an optimal rank-decomposition of the given graph.

This function is available as a method of the *Graph* class. See *rank_decomposition*.

INPUT:

- `verbose` – boolean (default: `False`); whether to display progress information while computing the decomposition

OUTPUT:

A pair `(rankwidth, decomposition_tree)`, where `rankwidth` is a numerical value and `decomposition_tree` is a ternary tree describing the decomposition (cf. the module's documentation).

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.rankwidth import rank_decomposition
sage: g = graphs.PetersenGraph()
sage: rank_decomposition(g)
(3, Graph on 19 vertices)
```

On more than 32 vertices:

```
sage: g = graphs.RandomGNP(40, .5)
sage: rank_decomposition(g)
Traceback (most recent call last):
...
RuntimeError: the rank decomposition cannot be computed on graphs of >= 32␣
↪vertices
```

The empty graph:

```
sage: g = Graph()
sage: rank_decomposition(g)
(0, Graph on 0 vertices)
```

5.21 Bandwidth of undirected graphs

5.21.1 Definition

The bandwidth $bw(M)$ of a matrix M is the smallest integer k such that all non-zero entries of M are at distance k from the diagonal. The bandwidth $bw(G)$ of an undirected graph G is the minimum bandwidth of the adjacency matrix of G, over all possible relabellings of its vertices.

Path spanner: alternatively, the bandwidth measures how tightly a path represents the distance of a graph G. Indeed, if the vertices of G can be ordered as $v_1, ..., v_n$ in such a way that $k \times d_G(v_i, v_j) \geq |i - j|$ then $bw(G) \leq k$.

> **Proof:** for all $v_i \sim v_j$ (i.e. $d_G(v_i, v_j) = 1$), the constraint ensures that $k \geq |i - j|$, meaning that adjacent vertices are at distance at most k in the path ordering. That alone is sufficient to ensure that $bw(G) \leq k$.
>
> As a byproduct, we obtain that $k \times d_G(v_i, v_j) \geq |i - j|$ in general: let $v_{s_0}, ..., v_{s_i}$ be the vertices of a shortest (v_i, v_j)-path. We have:
>
> $$\begin{aligned} k \times d_G(v_i, v_j) &= k \times d_G(v_i, v_{s_0}) + k \times d_G(v_{s_0}, v_{s_1}) + ... + k \times d_G(v_{s_{i-1}}, v_{s_i}) + k \times d_G(v_{s_i}, v_j) \\ &\geq |v_i - v_{s_0}| + |v_{s_0} - v_{s_1}| + ... + |v_{s_{i-1}} - v_{s_i}| + |v_{s_i} - v_j| \\ &\geq |v_i - v_j| \end{aligned}$$

5.21.2 Satisfiability of a partial assignment

Let us suppose that the first i vertices $v_1, ..., v_i$ of G have already been assigned positions $p_1, ..., p_i$ in an ordering of $V(G)$ of bandwidth $\leq k$. Where can v_{i+1} appear ?

Because of the previous definition, p_{i+1} must be at distance at most $k \times d_G(v_1, v_{i+1})$ from p_1, and in general at distance at most $k \times d_G(v_j, v_{i+1})$ from p_j. Each range is an interval of $\{1, ..., n\} \backslash \{p_1, ..., p_i\}$, and because the intersection of two intervals is again an interval we deduce that in order to satisfy all these constraints simultaneously p_j must belong to an interval defined from this partial assignment.

Applying this rule to all non-assigned vertices, we deduce that each of them must be assigned to a given interval of $\{1, ..., n\}$. Note that this can also be extended to the already assigned vertices, by saying that v_j with $j < i$ must be assigned within the interval $[p_j, p_j]$.

This problem is not always satisfiable, e.g. 5 vertices cannot all be assigned to the elements of $[10, 13]$. This is a matching problem which, because all admissible sets are intervals, can be solved quickly.

5.21.3 Solving the matching problem

Let n points $v_1, ..., v_n$ be given, along with two functions $m, M : [n] \mapsto [n]$. Is there an ordering $p_1, ..., p_n$ of them such that $m(v_i) \leq p_i \leq M(v_i)$? This is equivalent to Hall's bipartite matching theorem, and can in this specific case be solved by the following algorithm:

- Consider all vertices v sorted increasingly according to $M(v)$
- For each of them, assign to v the smallest position in $[m(v), M(v)]$ which has not been assigned yet. If there is none, the assignment problem is not satisfiable.

Note that the latest operation can be performed with very few bitset operations (provided that $n < 64$).

5.21.4 The algorithm

This section contains totally subjective choices, that may be changed in the hope to get better performances.

- Try to find a satisfiable ordering by filling positions, one after the other (and not by trying to find each vertex' position)
- Fill the positions in this order: $0, n-1, 1, n-2, 3, n-3, ...$

Note: There is some symmetry to break as the reverse of a satisfiable ordering is also a satisfiable ordering.

5.21.5 This module contains the following methods

bandwidth()	Compute the bandwidth of an undirected graph
bandwidth_heuristics()	Use Boost heuristics to approximate the bandwidth of the input graph

5.21.6 Functions

sage.graphs.graph_decompositions.bandwidth.**bandwidth**(*G*, *k=None*)

Compute the bandwidth of an undirected graph.

For a definition of the bandwidth of a graph, see the documentation of the *bandwidth* module.

INPUT:

- `G` – a graph
- `k` – integer (default: `None`); set to an integer value to test whether $bw(G) \leq k$, or to `None` (default) to compute $bw(G)$

OUTPUT:

When k is an integer value, the function returns either `False` or an ordering of cost $\leq k$.

When k is equal to `None`, the function returns a pair `(bw, ordering)`.

See also:

sage.graphs.generic_graph.GenericGraph.adjacency_matrix() – return the adjacency matrix from an ordering of the vertices.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.bandwidth import bandwidth
sage: G = graphs.PetersenGraph()
sage: bandwidth(G,3)
False
sage: bandwidth(G)
(5, [0, 4, 5, 8, 1, 9, 3, 7, 6, 2])
sage: G.adjacency_matrix(vertices=[0, 4, 5, 8, 1, 9, 3, 7, 6, 2])
[0 1 1 0 1 0 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0]
[1 0 0 1 0 0 0 1 0 0]
[0 0 1 0 0 0 1 0 1 0]
[1 0 0 0 0 0 0 0 1 1]
[0 1 0 0 0 0 0 1 1 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 1 0 0 0 0 0 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 1 1 0 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
sage: G = graphs.ChvatalGraph()
sage: bandwidth(G)
(6, [0, 5, 9, 4, 10, 1, 6, 11, 3, 8, 7, 2])
sage: G.adjacency_matrix(vertices=[0, 5, 9, 4, 10, 1, 6, 11, 3, 8, 7, 2])
[0 0 1 1 0 1 1 0 0 0 0 0]
[0 0 0 1 1 1 0 1 0 0 0 0]
[1 0 0 0 1 0 0 1 1 0 0 0]
[1 1 0 0 0 0 0 0 1 1 0 0]
[0 1 1 0 0 0 1 0 0 1 0 0]
[1 1 0 0 0 0 0 0 0 0 1 1]
[1 0 0 0 1 0 0 1 0 0 0 1]
[0 1 1 0 0 0 1 0 0 0 1 0]
[0 0 1 1 0 0 0 0 0 0 1 1]
[0 0 0 1 1 0 0 0 0 0 1 1]
[0 0 0 0 0 1 0 1 1 1 0 0]
[0 0 0 0 0 1 1 0 1 1 0 0]
```

5.22 Cutwidth

This module implements several algorithms to compute the cutwidth of a graph and the corresponding ordering of the vertices. It also implements tests functions for evaluation the width of a linear ordering (or layout).

Given an ordering $v_1, \cdots, v_n$ of the vertices of $V(G)$, its *cost* is defined as:

$$c(v_1, ..., v_n) = \max_{1 \leq i \leq n-1} c'(\{v_1, ..., v_i\})$$

Where

$$c'(S) = |\{(u, w) \in E(G) \mid u \in S \text{ and } w \in V(G) \backslash S\}|$$

The *cutwidth* of a graph G is equal to the minimum cost of an ordering of its vertices.

This module contains the following methods

cutwidth()	Return the cutwidth of the graph and the corresponding vertex ordering.
cutwidth_dyn()	Compute the cutwidth of G using an exponential time and space algorithm based on dynamic programming
cutwidth_MILP()	Compute the cutwidth of G and the optimal ordering of its vertices using an MILP formulation
width_of_cut_decomposition()	Return the width of the cut decomposition induced by the linear ordering L of the vertices of G

5.22.1 Exponential algorithm for cutwidth

In order to find an optimal ordering of the vertices for the vertex separation, this algorithm tries to save time by computing the function $c'(S)$ **at most once** once for each of the sets $S \subseteq V(G)$. These values are stored in an array of size 2^n where reading the value of $c'(S)$ or updating it can be done in constant time.

Assuming that we can compute the cost of a set S and remember it, finding an optimal ordering is an easy task. Indeed, we can think of the sequence $v_1, ..., v_n$ of vertices as a sequence of *sets* $\{v_1\}, \{v_1, v_2\}, ..., \{v_1, ..., v_n\}$, whose cost is precisely $\max c'(\{v_1\}), c'(\{v_1, v_2\}), ..., c'(\{v_1, ..., v_n\})$. Hence, when considering the digraph on the 2^n sets $S \subseteq V(G)$ where there is an arc from S to S' if $S' = S \cup \{v\}$ for some v (that is, if the sets S and S' can be consecutive in a sequence), an ordering of the vertices of G corresponds to a *path* from $\emptyset$ to $\{v_1, ..., v_n\}$. In this setting, checking whether there exists a ordering of cost less than k can be achieved by checking whether there exists a directed path $\emptyset$ to $\{v_1, ..., v_n\}$ using only sets of cost less than k. This is just a depth-first-search, for each k.

Lazy evaluation of c'

In the previous algorithm, most of the time is actually spent on the computation of $c'(S)$ for each set $S \subseteq V(G)$ – i.e. 2^n computations of neighborhoods. This can be seen as a huge waste of time when noticing that it is useless to know that the value $c'(S)$ for a set S is less than k if all the paths leading to S have a cost greater than k. For this reason, the value of $c'(S)$ is computed lazily during the depth-first search. Explanation :

When the depth-first search discovers a set of size less than k, the costs of its out-neighbors (the potential sets that could follow it in the optimal ordering) are evaluated. When an out-neighbor is found that has a cost smaller than k, the depth-first search continues with this set, which is explored with the hope that it could lead to a path toward $\{v_1, ..., v_n\}$. On the other hand, if an out-neighbour has a cost larger than k it is useless to attempt to build a cheap sequence going though this set, and the exploration stops there. This way, a large number of sets will never be evaluated and *a lot* of computational time is saved this way.

Besides, some improvement is also made by "improving" the values found by c'. Indeed, $c'(S)$ is a lower bound on the cost of a sequence containing the set S, but if all out-neighbors of S have a cost of $c'(S) + 5$ then one knows that having S in a sequence means a total cost of at least $c'(S) + 5$. For this reason, for each set S we store the value of $c'(S)$, and replace it by $\max(c'(S), \min_{\text{next}})$ (where $\min_{\text{next}}$ is the minimum of the costs of the out-neighbors of S) once the costs of these out-neighbors have been evaluated by the algorithm.

This algorithm and its implementation are very similar to *sage.graphs.graph_decompositions.vertex_separation.vertex_separation_exp()*. The main difference is in the computation of $c'(S)$. See the *vertex separation module's documentation* for more details on this algorithm.

Note: Because of its current implementation, this algorithm only works on graphs on strictly less than 32 vertices. This can be changed to 64 if necessary, but 32 vertices already require 4GB of memory.

5.22.2 MILP formulation for the cutwidth

We describe a mixed integer linear program (MILP) for determining an optimal layout for the cutwidth of G.

Variables:

- x_v^k – Variable set to 1 if vertex v is placed in the ordering at position i with $i \leq k$, and 0 otherwise.
- $y_{u,v}^k$ – Variable set to 1 if one of u or v is at a position $i \leq k$ and the other is at a position $j > k$, and so we have to count edge uv at position k. Otherwise, $y_{u,v}^k = 0$. The value of $y_{u,v}^k$ is a xor of the values of x_u^k and x_v^k.
- z – Objective value to minimize. It is equal to the maximum over all position k of the number of edges with one extremity at position at most k and the other at position stricly more than k, that is $\sum_{uv \in E} y_{u,v}^k$.

MILP formulation:

Minimize:

$$z$$

Subject to:

$$\begin{aligned}
\sum_{i=0}^{k-1} x_v^i \leq k * x_v^k \qquad & \forall v \in V,\ k \in [1, n-1] \quad (1)\\
x_v^n = 1 \qquad & \forall v \in V \quad (2)\\
\sum_{v \in V} x_v^k = k+1 \qquad & \forall k \in [0, n-1] \quad (3)\\
x_u^k - x_v^k \leq y_{u,v}^k \qquad & \forall uv \in E,\ \forall k \in [0, n-1] \quad (4)\\
x_v^k - x_u^k \leq y_{u,v}^k \qquad & \forall uv \in E,\ \forall k \in [0, n-1] \quad (5)\\
\sum_{uv \in E} y_{u,v}^k \leq z \qquad & \forall k \in [0, n-1] \quad (6)\\
0 \leq z \leq |E| &
\end{aligned}$$

Constraints (1)-(3) ensure that all vertices have a distinct position. Constraints (4)-(5) force variable $y_{u,v}^k$ to 1 if the edge is in the cut. Constraint (6) count the number of edges starting at position at most k and ending at a position stricly larger than k.

This formulation corresponds to method *cutwidth_MILP()*.

5.22.3 Authors

- David Coudert (2015-06): Initial version

5.22.4 Methods

sage.graphs.graph_decompositions.cutwidth.**cutwidth**(*G*, *algorithm='exponential'*, *cut_off=0*, *solver=None*, *verbose=False*)

Return the cutwidth of the graph and the corresponding vertex ordering.

INPUT:

- `G` – a Graph or a DiGraph
- `algorithm` – string (default: `"exponential"`); algorithm to use among:
  - `exponential` – Use an exponential time and space algorithm based on dynamic programming. This algorithm only works on graphs with strictly less than 32 vertices.
  - `MILP` – Use a mixed integer linear programming formulation. This algorithm has no size restriction but could take a very long time.
- `cut_off` – integer (default: 0); used to stop the search as soon as a solution with width at most `cut_off` is found, if any. If this bound cannot be reached, the best solution found is returned.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. This parameter is used only when `algorithm='MILP'`. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – boolean (default: `False`); whether to display information on the computations.

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

EXAMPLES:

Cutwidth of a Complete Graph:

```
sage: from sage.graphs.graph_decompositions.cutwidth import cutwidth
sage: G = graphs.CompleteGraph(5)
sage: cw,L = cutwidth(G); cw
6
sage: K = graphs.CompleteGraph(6)
sage: cw,L = cutwidth(K); cw
9
sage: cw,L = cutwidth(K+K); cw
9
```

The cutwidth of a $p \times q$ Grid Graph with $p \leq q$ is $p + 1$:

```
sage: from sage.graphs.graph_decompositions.cutwidth import cutwidth
sage: G = graphs.Grid2dGraph(3,3)
sage: cw,L = cutwidth(G); cw
4
sage: G = graphs.Grid2dGraph(3,5)
sage: cw,L = cutwidth(G); cw
4
```

sage.graphs.graph_decompositions.cutwidth.**cutwidth_MILP**(*G*, *lower_bound=0*, *solver=None*, *verbose=0*)

MILP formulation for the cutwidth of a Graph.

This method uses a mixed integer linear program (MILP) for determining an optimal layout for the cutwidth of G. See the *module's documentation* for more details on this MILP formulation.

INPUT:

- `G` – a Graph
- `lower_bound` – integer (default: 0); the algorithm searches for a solution with cost larger or equal to `lower_bound`. If the given bound is larger than the optimal solution the returned solution might not be optimal. If the given bound is too high, the algorithm might not be able to find a feasible solution.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

EXAMPLES:

Cutwidth of a Cycle graph:

```
sage: from sage.graphs.graph_decompositions import cutwidth
sage: G = graphs.CycleGraph(5)
sage: cw, L = cutwidth.cutwidth_MILP(G); cw
2
sage: cw == cutwidth.width_of_cut_decomposition(G, L)
True
sage: cwe, Le = cutwidth.cutwidth_dyn(G); cwe
2
```

Cutwidth of a Complete graph:

```
sage: from sage.graphs.graph_decompositions import cutwidth
sage: G = graphs.CompleteGraph(4)
sage: cw, L = cutwidth.cutwidth_MILP(G); cw
4
sage: cw == cutwidth.width_of_cut_decomposition(G, L)
True
```

Cutwidth of a Path graph:

```
sage: from sage.graphs.graph_decompositions import cutwidth
sage: G = graphs.PathGraph(3)
sage: cw, L = cutwidth.cutwidth_MILP(G); cw
1
sage: cw == cutwidth.width_of_cut_decomposition(G, L)
True
```

sage.graphs.graph_decompositions.cutwidth.**cutwidth_dyn**(*G*, *lower_bound=0*)

Dynamic programming algorithm for the cutwidth of a Graph.

This function uses dynamic programming algorithm for determining an optimal layout for the cutwidth of G. See the *module's documentation* for more details on this method.

INPUT:

- `G` – a Graph
- `lower_bound` – integer (default: 0); the algorithm returns immediately if it finds a solution lower or equal to `lower_bound` (in which case it may not be optimal).

OUTPUT:

A pair `(cost, ordering)` representing the optimal ordering of the vertices and its cost.

Note: Because of its current implementation, this algorithm only works on graphs on strictly less than 32 vertices. This can be changed to 63 if necessary, but 32 vertices already require 4GB of memory.

---

sage.graphs.graph_decompositions.cutwidth.**width_of_cut_decomposition**(*G*, *L*)

Return the width of the cut decomposition induced by the linear ordering L of the vertices of G.

If G is an instance of *Graph*, this function returns the width $cw_L(G)$ of the cut decomposition induced by the linear ordering L of the vertices of G.

$$cw_L(G) = \max_{0 \leq i < |V|-1} |\{(u,w) \in E(G) \mid u \in L[:i] \text{ and } w \in V(G) \setminus L[:i]\}|$$

INPUT:

- `G` – a Graph
- `L` – a linear ordering of the vertices of `G`

EXAMPLES:

Cut decomposition of a Cycle graph:

```
sage: from sage.graphs.graph_decompositions import cutwidth
sage: G = graphs.CycleGraph(6)
sage: L = G.vertices()
sage: cutwidth.width_of_cut_decomposition(G, L)
2
```

Cut decomposition of a Path graph:

```
sage: from sage.graphs.graph_decompositions import cutwidth
sage: P = graphs.PathGraph(6)
sage: cutwidth.width_of_cut_decomposition(P, [0, 1, 2, 3, 4, 5])
1
sage: cutwidth.width_of_cut_decomposition(P, [5, 0, 1, 2, 3, 4])
2
sage: cutwidth.width_of_cut_decomposition(P, [0, 2, 4, 1, 3, 5])
5
```

5.23 Products of graphs

This module gathers everything related to graph products. At the moment it contains an implementation of a recognition algorithm for graphs that can be written as a Cartesian product of smaller ones.

Author:

- Nathann Cohen (May 2012 – coded while watching the election of Francois Hollande on TV)

5.23.1 Cartesian product of graphs – the recognition problem

First, a definition:

Definition The Cartesian product of two graphs G and H, denoted $G\Box H$, is a graph defined on the pairs $(g,h) \in V(G) \times V(H)$.

Two elements $(g,h),(g',h') \in V(G\Box H)$ are adjacent in $G\Box H$ if and only if :

- $g = g'$ and $hh' \in H$; or
- $h = h'$ and $gg' \in G$

Two remarks follow :

1. The Cartesian product is commutative
2. Any edge uv of a graph $G_1\Box \cdots \Box G_k$ can be given a color i corresponding to the unique index i such that u_i and v_i differ.

The problem that is of interest to us in the present module is the following:

Recognition problem Given a graph G, can we guess whether there exist graphs $G_1, ..., G_k$ such that $G = G_1\Box \cdots \Box G_k$?

This problem can actually be solved, and the resulting factorization is unique. What is explained below can be found in the book *Handbook of Product Graphs* [HIK2011].

Everything is actually based on simple observations. Given a graph G, finding out whether G can be written as the product of several graphs can be attempted by trying to color its edges according to some rules. Indeed, if we are to color the edges of G in such a way that each color class represents a factor of G, we must ensure several things.

Remark 1 In any cycle of G no color can appear exactly once.

Indeed, if only one edge uv of a cycle were labelled with color i, it would mean that:

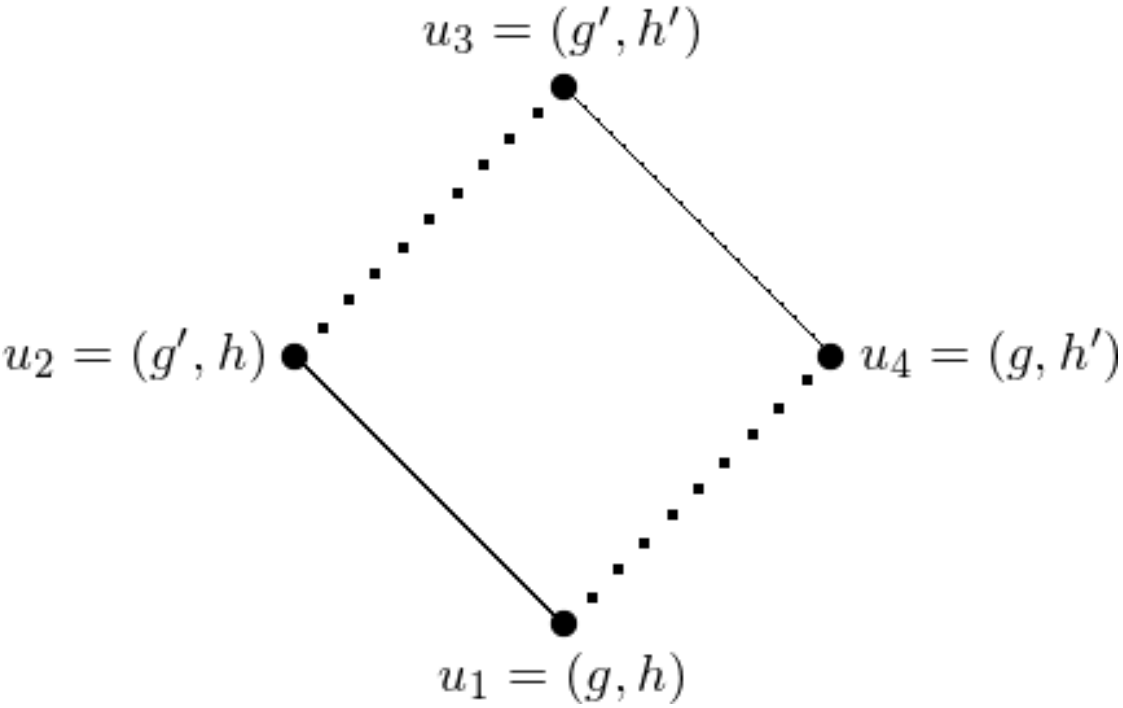
1. The only difference between u and v lies in their i th coordinate
2. It is possible to go from u to v by changing only coordinates different from the i th

A contradiction indeed.

That means that, for instance, the edges of a triangle necessarily have the same color.

Remark 2 If two consecutive edges u_1u_2 and u_2u_3 have different colors, there necessarily exists a unique vertex u_4 different from u_2 and incident to both u_1 and u_3.

In this situation, opposed edges necessarily have the same colors because of the previous remark.



1st criterion : As a corollary, we know that:

1. If two vertices u, v have a *unique* common neighbor x, then ux and xv have the same color.
2. If two vertices u, v have more that two common neighbors $x_1, ..., x_k$ then all edges between the x_i and the vertices of u, v have the same color. This is also a consequence of the first remark.

2nd criterion : if two edges uv and $u'v'$ of the product graph $G\Box H$ are such that $d(u, u') + d(v, v') \neq d(u, v') + d(v, u')$ then the two edges uv and $u'v'$ necessarily have the same color.

> This is a consequence of the fact that for any two vertices u, v of $G\Box H$ (where $u = (u_G, u_H)$ and $v = (v_G, v_H)$), we have $d(u, v) = d_G(u_G, v_G) + d_H(u_H, v_H)$. Indeed, a shortest path

from u to v in $G\square H$ contains the information of a shortest path from u_G to v_G in G, and a shortest path from u_H to v_H in H.

The algorithm

The previous remarks tell us that some edges are in some way equivalent to some others, i.e. that their colors are equal. In order to compute the coloring we are looking for, we therefore build a graph on the *edges* of a graph G, linking two edges whenever they are found to be equivalent according to the previous remarks.

All that is left to do is to compute the connected components of this new graph, as each of them representing the edges of a factor. Of course, only one connected component indicates that the graph has no factorization.

Then again, please refer to [HIK2011] for any technical question.

To Do

This implementation is made at Python level, and some parts of the algorithm could be rewritten in Cython to save time. Especially when enumerating all pairs of edges and computing their distances. This can easily be done in C with the functions from the *sage.graphs.distances_all_pairs* module.

5.23.2 Methods

sage.graphs.graph_decompositions.graph_products.**is_cartesian_product**(*g*, *certificate=False*, *relabeling=False*)

Test whether the graph is a Cartesian product.

INPUT:

- `certificate` – boolean (default: `False`); if `certificate = False` (default) the method only returns `True` or `False` answers. If `certificate = True`, the `True` answers are replaced by the list of the factors of the graph.
- `relabeling` – boolean (default: `False`); if `relabeling = True` (implies `certificate = True`), the method also returns a dictionary associating to each vertex its natural coordinates as a vertex of a product graph. If g is not a Cartesian product, `None` is returned instead.

See also:

- *sage.graphs.generic_graph.GenericGraph.cartesian_product()*
- *graph_products* – a module on graph products.

Note: This algorithm may run faster whenever the graph's vertices are integers (see *relabel()*). Give it a try if it is too slow !

EXAMPLES:

The Petersen graph is prime:

```
sage: from sage.graphs.graph_decompositions.graph_products import is_cartesian_
↪product
sage: g = graphs.PetersenGraph()
sage: is_cartesian_product(g)
False
```

A 2d grid is the product of paths:

```
sage: g = graphs.Grid2dGraph(5,5)
sage: p1, p2 = is_cartesian_product(g, certificate = True)
sage: p1.is_isomorphic(graphs.PathGraph(5))
True
sage: p2.is_isomorphic(graphs.PathGraph(5))
True
```

Forgetting the graph's labels, then finding them back:

```
sage: g.relabel()
sage: b,D = g.is_cartesian_product(g, relabeling=True)
sage: b
True
sage: D  # random isomorphism
{0: (20, 0), 1: (20, 1), 2: (20, 2), 3: (20, 3), 4: (20, 4),
 5: (15, 0), 6: (15, 1), 7: (15, 2), 8: (15, 3), 9: (15, 4),
 10: (10, 0), 11: (10, 1), 12: (10, 2), 13: (10, 3), 14: (10, 4),
 15: (5, 0), 16: (5, 1), 17: (5, 2), 18: (5, 3), 19: (5, 4),
 20: (0, 0), 21: (0, 1), 22: (0, 2), 23: (0, 3), 24: (0, 4)}
```

And of course, we find the factors back when we build a graph from a product:

```
sage: g = graphs.PetersenGraph().cartesian_product(graphs.CycleGraph(3))
sage: g1, g2 = is_cartesian_product(g, certificate = True)
sage: any( x.is_isomorphic(graphs.PetersenGraph()) for x in [g1,g2])
True
sage: any( x.is_isomorphic(graphs.CycleGraph(3)) for x in [g1,g2])
True
```

5.24 Modular Decomposition

This module implements the function for computing the modular decomposition of undirected graphs.

class sage.graphs.graph_decompositions.modular_decomposition.**Node**(*node_type*)
Bases: object

Node class stores information about the node type, node split and index of the node in the parent tree.

Node type can be PRIME, SERIES, PARALLEL, NORMAL or FOREST. Node split can be NO_SPLIT, LEFT_SPLIT, RIGHT_SPLIT or BOTH_SPLIT. A node is split in the refinement phase and the split used is propagated to the ancestors.

- node_type – is of type NodeType and specifies the type of node
- node_split – is of type NodeSplit and specifies the type of splits which have occurred in the node and its descendants
- index_in_root – specifies the index of the node in the forest obtained after promotion phase
- comp_num – specifies the number given to nodes in a (co)component before refinement

- `is_separated` – specifies whether a split has occurred with the node as the root

has_left_split()
Check whether `self` has `LEFT_SPLIT`.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.LEFT_SPLIT)
sage: node.has_left_split()
True
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.BOTH_SPLIT)
sage: node.has_left_split()
True
```

has_right_split()
Check whether `self` has `RIGHT_SPLIT`.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.RIGHT_SPLIT)
sage: node.has_right_split()
True
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.BOTH_SPLIT)
sage: node.has_right_split()
True
```

set_node_split(*node_split*)
Add node_split to the node split of self.

`LEFT_SPLIT` and `RIGHT_SPLIT` can exist together in `self` as `BOTH_SPLIT`.

INPUT:

- `node_split` – node_split to be added to self

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.LEFT_SPLIT)
sage: node.node_split == NodeSplit.LEFT_SPLIT
True
sage: node.set_node_split(NodeSplit.RIGHT_SPLIT)
sage: node.node_split == NodeSplit.BOTH_SPLIT
True
sage: node = Node(NodeType.PRIME)
sage: node.set_node_split(NodeSplit.BOTH_SPLIT)
sage: node.node_split == NodeSplit.BOTH_SPLIT
True
```

class sage.graphs.graph_decompositions.modular_decomposition.**NodeSplit**
Bases: `enum.Enum`

Enumeration class used to specify the split that has occurred at the node or at any of its descendants.

NodeSplit is defined for every node in modular decomposition tree and is required during the refinement and promotion phase of modular decomposition tree computation. Various node splits defined are

- LEFT_SPLIT – indicates a left split has occurred
- RIGHT_SPLIT – indicates a right split has occurred
- BOTH_SPLIT – indicates both left and right split have occurred
- NO_SPLIT – indicates no split has occurred

class sage.graphs.graph_decompositions.modular_decomposition.**NodeType**
Bases: enum.Enum

NodeType is an enumeration class used to define the various types of nodes in modular decomposition tree.

The various node types defined are

- PARALLEL – indicates the node is a parallel module
- SERIES – indicates the node is a series module
- PRIME – indicates the node is a prime module
- FOREST – indicates a forest containing trees
- NORMAL – indicates the node is normal containing a vertex

class sage.graphs.graph_decompositions.modular_decomposition.**VertexPosition**
Bases: enum.Enum

Enumeration class used to define position of a vertex w.r.t source in modular decomposition.

For computing modular decomposition of connected graphs a source vertex is chosen. The position of vertex is w.r.t this source vertex. The various positions defined are

- LEFT_OF_SOURCE – indicates vertex is to left of source and is a neighbour of source vertex
- RIGHT_OF_SOURCE – indicates vertex is to right of source and is connected to but not a neighbour of source vertex
- SOURCE – indicates vertex is source vertex

sage.graphs.graph_decompositions.modular_decomposition.**assembly**(*graph*, *root*, *vertex_status*, *vertex_dist*)

Assemble the forest obtained after the promotion phase into a modular decomposition tree.

INPUT:

- graph – graph whose MD tree is to be computed
- root – Forest which would be assembled into a MD tree
- vertex_status – Dictionary which stores the position of vertex with respect to the source
- vertex_dist – Dictionary which stores the distance of vertex from source vertex

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
```

(continues on next page)

(continued from previous page)

```
sage: g.add_edge(2, 7)
sage: g.add_edge(6, 1)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: forest.children[0].comp_num = 1
sage: forest.children[1].comp_num = 1
sage: forest.children[1].children[0].comp_num = 1
sage: forest.children[1].children[1].comp_num = 1
sage: number_components(forest, vertex_status)
sage: assembly(g, forest, vertex_status, vertex_dist)
sage: forest.children
[PRIME [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        PARALLEL [NORMAL [6], NORMAL [7]], NORMAL [1]]]

sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: number_cocomponents(forest, vertex_status)
sage: assembly(g, forest, vertex_status, vertex_dist)
sage: forest.children
[PRIME [NORMAL [2], SERIES [NORMAL [4], NORMAL [5], NORMAL [3]],
        PARALLEL [NORMAL [6], NORMAL [7]], NORMAL [1]]]
```

sage.graphs.graph_decompositions.modular_decomposition.**check_parallel**(*graph*, *root*, *left*, *right*, *source_index*, *mu*, *vertex_dist*, *vertices_in_component*)

Assemble the forest to create a parallel module.

INPUT:

- `root` – forest which needs to be assembled
- `left` – the leftmost fragment of the last module
- `right` – the rightmost fragment of the last module
- `source_index` – index of the tree containing the source vertex
- `mu` – dictionary which maps the (co)components with their mu values
- `vertex_dist` – dictionary which stores the distance of vertex from source vertex
- `vertices_in_component` – dictionary which stores a list of various vertices in a (co)component

OUTPUT:

`[module_formed, source_index]` where `module_formed` is `True` if module is formed else `False` and `source_index` is the index of the new module which contains the source vertex

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(4, 1)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                    create_normal_node(7), create_normal_node(1)]
sage: forest.children.insert(1, series_node)
sage: forest.children.append(parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
```

(continues on next page)

(continued from previous page)

```
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 2}
sage: source_index = 2
sage: vertices_in_component = {}
sage: mu = {}
sage: left = right = forest.children[2]
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
....:     component.index_in_root = index
sage: for index, component in enumerate(forest.children):
....:     if index < source_index:
....:         mu[index] = compute_mu_for_co_component(g, index,
....:                                                 source_index, forest,
....:                                                 vertices_in_component)
....:     elif index > source_index:
....:         mu[index] = compute_mu_for_component(g, index,
....:                                              source_index, forest,
....:                                              vertices_in_component)
sage: number_components(forest, vertex_status)
sage: check_parallel(g, forest, left, right,
....:               source_index, mu, vertex_dist,
....:               vertices_in_component)
[True, 2]
sage: forest.children
[NORMAL [2],
 SERIES [NORMAL [4], NORMAL [5]],
 PARALLEL [NORMAL [3], NORMAL [6], NORMAL [7], NORMAL [1]]]
```

sage.graphs.graph_decompositions.modular_decomposition.**check_prime**(*graph*, *root*, *left*, *right*, *source_index*, *mu*, *vertex_dist*, *vertices_in_component*)

Assemble the forest to create a prime module.

INPUT:

- `root` – forest which needs to be assembled
- `left` – the leftmost fragment of the last module
- `right` – the rightmost fragment of the last module
- `source_index` – index of the tree containing the source vertex
- `mu` – dictionary which maps the (co)components with their mu values
- `vertex_dist` – dictionary which stores the distance of vertex from source vertex
- `vertices_in_component` – dictionary which stores a list of various vertices in a (co)component

OUTPUT:

`[module_formed, source_index]` where `module_formed` is `True` if module is formed else `False` and `source_index` is the index of the new module which contains the source vertex

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(6, 1)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: source_index = 2
sage: vertices_in_component = {}
sage: mu = {}
sage: left = right = forest.children[2]
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
....:     component.index_in_root = index
sage: for index, component in enumerate(forest.children):
....:     if index < source_index:
....:         mu[index] = compute_mu_for_co_component(g, index,
....:                                                 source_index, forest,
....:                                                 vertices_in_component)
....:     elif index > source_index:
....:         mu[index] = compute_mu_for_component(g, index,
....:                                              source_index, forest,
....:                                              vertices_in_component)
sage: forest.children[0].comp_num = 1
sage: forest.children[1].comp_num = 1
sage: forest.children[1].children[0].comp_num = 1
sage: forest.children[1].children[1].comp_num = 1
sage: number_components(forest, vertex_status)
sage: check_prime(g, forest, left, right,
....:             source_index, mu, vertex_dist,
....:             vertices_in_component)
[True, 0]
sage: forest.children
[PRIME [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        PARALLEL [NORMAL [6], NORMAL [7]], NORMAL [1]]]
```

sage.graphs.graph_decompositions.modular_decomposition.**check_series**(*root*, *left*, *right*, *source_index*, *mu*)

Assemble the forest to create a series module.

INPUT:

- `root` – forest which needs to be assembled
- `left` – The leftmost fragment of the last module
- `right` – The rightmost fragment of the last module
- `source_index` – index of the tree containing the source vertex
- `mu` – dictionary which maps the (co)components with their mu values
- `vertex_dist` – dictionary which stores the distance of vertex from source vertex
- `vertices_in_component` – dictionary which stores a list of various vertices in a (co)component

OUTPUT:

`[module_formed, source_index]` where `module_formed` is `True` if module is formed else `False` and `source_index` is the index of the new module which contains the source vertex

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: source_index = 2
sage: vertices_in_component = {}
```

(continues on next page)

(continued from previous page)

```
sage: mu = {}
sage: left = right = forest.children[2]
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
....:     component.index_in_root = index
sage: for index, component in enumerate(forest.children):
....:     if index < source_index:
....:         mu[index] = compute_mu_for_co_component(g, index,
....:                                                 source_index, forest,
....:                                                 vertices_in_component)
....:     elif index > source_index:
....:         mu[index] = compute_mu_for_component(g, index,
....:                                              source_index, forest,
....:                                              vertices_in_component)
sage: number_cocomponents(forest, vertex_status)
sage: number_components(forest, vertex_status)
sage: check_series(forest, left, right,
....:              source_index, mu)
[True, 1]
sage: forest.children
[NORMAL [2],
 SERIES [NORMAL [4], NORMAL [5], NORMAL [3]],
 PARALLEL [NORMAL [6], NORMAL [7]],
 NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**children_node_type**(*module*, *node_type*)

Check whether the node type of the childrens of `module` is `node_type`.

INPUT:

- `module` – module which is tested
- `node_type` – input node_type

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.OctahedralGraph()
sage: tree_root = modular_decomposition(g)
sage: print_md_tree(modular_decomposition(g))
SERIES
      PARALLEL
        2
        3
      PARALLEL
        1
        4
      PARALLEL
        0
        5
sage: children_node_type(tree_root, NodeType.SERIES)
False
sage: children_node_type(tree_root, NodeType.PARALLEL)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**clear_node_split_info**(*root*)

Set the node_split of nodes to NO_SPLIT

INPUT:

- root – the forest which needs to be cleared of split information

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: series_node.children[0].node_split = NodeSplit.LEFT_SPLIT
sage: series_node.node_split = NodeSplit.RIGHT_SPLIT
sage: forest.children.insert(1, series_node)
sage: clear_node_split_info(forest)
sage: series_node.node_split == NodeSplit.NO_SPLIT
True
sage: series_node.children[0].node_split == NodeSplit.NO_SPLIT
True
```

sage.graphs.graph_decompositions.modular_decomposition.**compute_mu_for_co_component**(*graph*, *component_index*, *source_index*, *root*, *vertices_in_co*

Compute the mu value for co-component

INPUT:

- graph – Graph whose MD tree needs to be computed
- component_index – index of the co-component
- source_index – index of the source in the forest
- root – the forest which needs to be assembled into a MD tree
- vertices_in_component – dictionary which maps index i to list of vertices in the tree at index i in the forest

OUTPUT:

The mu value (component in the forest) for the co-component

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
```

(continues on next page)

(continued from previous page)

```
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertices_in_component = {}
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
sage: compute_mu_for_co_component(g, 0, 2, forest,
....:                             vertices_in_component)
NORMAL [1]
sage: compute_mu_for_co_component(g, 1, 2, forest,
....:                             vertices_in_component)
NORMAL [3]
```

sage.graphs.graph_decompositions.modular_decomposition.**compute_mu_for_component**(*graph*, *component_index*, *source_index*, *root*, *vertices_in_compo*

Compute the mu value for component

INPUT:

- `graph` – Graph whose MD tree needs to be computed
- `component_index` – index of the component
- `source_index` – index of the source in the forest
- `root` – the forest which needs to be assembled into a MD tree
- `vertices_in_component` – dictionary which maps index i to list of vertices in the tree at the index i in the forest

OUTPUT:

The mu value (co-component in the forest) for the component

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(6, 1)
```

(continues on next page)

(continued from previous page)

```
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertices_in_component = {}
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
sage: compute_mu_for_component(g, 3, 2, forest,
....:                          vertices_in_component)
SERIES [NORMAL [4], NORMAL [5]]
sage: compute_mu_for_component(g, 4, 2, forest,
....:                          vertices_in_component)
NORMAL [2]
```

sage.graphs.graph_decompositions.modular_decomposition.**create_normal_node**(*vertex*)

Return a normal node with no children

INPUT:

- `vertex` – vertex number

OUTPUT:

A node object representing the vertex with node_type set as NodeType.NORMAL

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import create_
↪normal_node
sage: node = create_normal_node(2)
sage: node
NORMAL [2]
```

sage.graphs.graph_decompositions.modular_decomposition.**create_parallel_node**()

Return a parallel node with no children

OUTPUT:

A node object with node_type set as NodeType.PARALLEL

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import create_
↪parallel_node
sage: node = create_parallel_node()
sage: node
PARALLEL []
```

sage.graphs.graph_decompositions.modular_decomposition.**create_prime_node**()

Return a prime node with no children

OUTPUT:

A node object with node_type set as NodeType.PRIME

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import create_
↪prime_node
sage: node = create_prime_node()
sage: node
PRIME []
```

sage.graphs.graph_decompositions.modular_decomposition.**create_series_node**()

Return a series node with no children

OUTPUT:

A node object with node_type set as NodeType.SERIES

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import create_
↪series_node
sage: node = create_series_node()
sage: node
SERIES []
```

sage.graphs.graph_decompositions.modular_decomposition.**either_connected_or_not_connected**(*v*, *ve*
tic
gr

Check whether v is connected or disconnected to all vertices in the module.

INPUT:

- v – vertex tested
- vertices_in_module – list containing vertices in the module
- graph – graph to which the vertices belong

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.OctahedralGraph()
sage: print_md_tree(modular_decomposition(g))
SERIES
      PARALLEL
        2
        3
      PARALLEL
        1
         4
      PARALLEL
        0
        5
sage: either_connected_or_not_connected(2, [1, 4], g)
True
sage: either_connected_or_not_connected(2, [3, 4], g)
False
```

sage.graphs.graph_decompositions.modular_decomposition.**equivalent_trees**(*root1*, *root2*)

Check that two modular decomposition trees are the same.

Verify that the structure of the trees is the same. Two leaves are equivalent if they represent the same vertex, two internal nodes are equivalent if they have the same nodes type and the same number of children and there is a matching between the children such that each pair of children is a pair of equivalent subtrees.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: t1 = nested_tuple_to_tree((NodeType.SERIES, 1, 2,
....:              (NodeType.PARALLEL, 3, 4)))
sage: t2 = nested_tuple_to_tree((NodeType.SERIES,
....:              (NodeType.PARALLEL, 4, 3), 2, 1))
sage: equivalent_trees(t1, t2)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**form_module**(*index*, *other_index*, *tree_root*, *graph*)

Forms a module out of the modules in the module pair.

Let M_1 and M_2 be the input modules. Let V be the set of vertices in these modules. Suppose x is a neighbor of subset of the vertices in V but not all the vertices and x does not belong to V. Then the set of modules also include the module which contains x. This process is repeated until a module is formed and the formed module if subset of V is returned.

INPUT:

- `index` – first module in the module pair
- `other_index` – second module in the module pair
- `tree_root` – modular decomposition tree which contains the modules in the module pair
- `graph` – graph whose modular decomposition tree is created

OUTPUT:

`[module_formed, vertices]` where `module_formed` is `True` if module is formed else `False` and `vertices` is a list of vertices included in the formed module

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.HexahedralGraph()
sage: tree_root = modular_decomposition(g)
sage: form_module(0, 2, tree_root, g)
[False, {0, 1, 2, 3, 4, 5, 6, 7}]
```

sage.graphs.graph_decompositions.modular_decomposition.**gamma_classes**(*graph*)

Partition the edges of the graph into Gamma classes.

Two distinct edges are Gamma related if they share a vertex but are not part of a triangle. A Gamma class of edges is a collection of edges such that any edge in the class can be reached from any other by a chain of Gamma related edges (that are also in the class).

The two important properties of the Gamma class

- The vertex set corresponding to a Gamma class is a module
- If the graph is not fragile (neither it or its complement is disconnected) then there is exactly one class that visits all the vertices of the graph, and this class consists of just the edges that connect the maximal strong modules of that graph.

EXAMPLES:

The gamma_classes of the octahedral graph are the three 4-cycles corresponding to the slices through the center of the octahedron:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import gamma_
↪classes
sage: g = graphs.OctahedralGraph()
sage: sorted(gamma_classes(g), key=str)
[frozenset({0, 1, 4, 5}), frozenset({0, 2, 3, 5}), frozenset({1, 2, 3, 4})]
```

sage.graphs.graph_decompositions.modular_decomposition.**get_child_splits**(*root*)
Add the node_split of children to the parent node

INPUT:

- `root` – input modular decomposition tree

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: series_node.children[0].node_split = NodeSplit.LEFT_SPLIT
sage: series_node.node_split = NodeSplit.RIGHT_SPLIT
sage: forest.children.insert(1, series_node)
sage: get_child_splits(forest)
sage: series_node.node_split == NodeSplit.BOTH_SPLIT
True
sage: forest.node_split == NodeSplit.BOTH_SPLIT
True
```

sage.graphs.graph_decompositions.modular_decomposition.**get_module_type**(*graph*)
Return the module type of the root of the modular decomposition tree of `graph`.

INPUT:

- `graph` – input sage graph

OUTPUT:

`PRIME` if graph is PRIME, `PARALLEL` if graph is PARALLEL and `SERIES` if graph is of type SERIES

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import get_
↪module_type
sage: g = graphs.HexahedralGraph()
sage: get_module_type(g)
PRIME
```

sage.graphs.graph_decompositions.modular_decomposition.**get_vertex_in**(*node*)
Return the first vertex encountered in the depth-first traversal of the tree rooted at node

INPUT:

- `tree` – input modular decomposition tree

OUTPUT:

Return the first vertex encountered in recursion

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: forest.children.insert(1, series_node)
sage: get_vertex_in(forest)
2
```

sage.graphs.graph_decompositions.modular_decomposition.**get_vertices**(*component_root*)

Compute the list of vertices in the (co)component

INPUT:

- component_root – root of the (co)component whose vertices need to be returned as a list

OUTPUT:

list of vertices in the (co)component

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: get_vertices(forest)
[2, 4, 5, 3, 6, 7, 1]
```

sage.graphs.graph_decompositions.modular_decomposition.**habib_maurer_algorithm**(*graph*, *g_classes=None*)

Compute the modular decomposition by the algorithm of Habib and Maurer

Compute the modular decomposition of the given graph by the algorithm of Habib and Maurer [HM1979] . If the graph is disconnected or its complement is disconnected return a tree with a PARALLEL or SERIES node at the root and children being the modular decomposition of the subgraphs induced by the components. Otherwise, the root is PRIME and the modules are identified by having identical neighborhoods in the gamma class that spans the vertices of the subgraph (exactly one is guaranteed to exist). The gamma classes only need to be computed once, as the algorithm computes the the classes for the current root and each of the submodules. See also [BM1983] for an equivalent algorithm described in greater detail.

INPUT:

- graph – the graph for which modular decomposition tree needs to be computed

- `g_classes` – dictionary (default: `None`); a dictionary whose values are the gamma classes of the graph, and whose keys are a frozenset of the vertices corresponding to the class. Used internally.

OUTPUT:

The modular decomposition tree of the graph.

EXAMPLES:

The Icosahedral graph is Prime:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: print_md_tree(habib_maurer_algorithm(graphs.IcosahedralGraph()))  # py2
PRIME
 8
 0
 1
 3
 7
 4
 5
 2
 10
 11
 9
 6
```

The Octahedral graph is not Prime:

```
sage: print_md_tree(habib_maurer_algorithm(graphs.OctahedralGraph()))
SERIES
 PARALLEL
  0
  5
 PARALLEL
  1
  4
 PARALLEL
  2
  3
```

Tetrahedral Graph is Series:

```
sage: print_md_tree(habib_maurer_algorithm(graphs.TetrahedralGraph()))
SERIES
 0
 1
 2
 3
```

Modular Decomposition tree containing both parallel and series modules:

```
sage: d = {2:[4,3,5], 1:[4,3,5], 5:[3,2,1,4], 3:[1,2,5], 4:[1,2,5]}
sage: g = Graph(d)
sage: print_md_tree(habib_maurer_algorithm(g))
SERIES
 PARALLEL
  1
  2
```

(continues on next page)

(continued from previous page)

```
 PARALLEL
  3
  4
 5
```

Graph from Marc Tedder implementation of modular decomposition:

```
sage: d = {1:[5,4,3,24,6,7,8,9,2,10,11,12,13,14,16,17], 2:[1],
....:       3:[24,9,1], 4:[5,24,9,1], 5:[4,24,9,1], 6:[7,8,9,1],
....:       7:[6,8,9,1], 8:[6,7,9,1], 9:[6,7,8,5,4,3,1], 10:[1],
....:       11:[12,1], 12:[11,1], 13:[14,16,17,1], 14:[13,17,1],
....:       16:[13,17,1], 17:[13,14,16,18,1], 18:[17], 24:[5,4,3,1]}
sage: g = Graph(d)
sage: test_modular_decomposition(habib_maurer_algorithm(g), g)
True
```

Graph from the Wikipedia article Modular_decomposition:

```
sage: d2 = {1:[2,3,4], 2:[1,4,5,6,7], 3:[1,4,5,6,7], 4:[1,2,3,5,6,7],
....:       5:[2,3,4,6,7], 6:[2,3,4,5,8,9,10,11],
....:       7:[2,3,4,5,8,9,10,11], 8:[6,7,9,10,11], 9:[6,7,8,10,11],
....:       10:[6,7,8,9], 11:[6,7,8,9]}
sage: g = Graph(d2)
sage: test_modular_decomposition(habib_maurer_algorithm(g), g)
True
```

Tetrahedral Graph is Series:

```
sage: print_md_tree(habib_maurer_algorithm(graphs.TetrahedralGraph()))
SERIES
 0
 1
 2
 3
```

Modular Decomposition tree containing both parallel and series modules:

```
sage: d = {2:[4,3,5], 1:[4,3,5], 5:[3,2,1,4], 3:[1,2,5], 4:[1,2,5]}
sage: g = Graph(d)
sage: print_md_tree(habib_maurer_algorithm(g))
SERIES
 PARALLEL
  1
  2
 PARALLEL
  3
  4
 5
```

sage.graphs.graph_decompositions.modular_decomposition.**has_left_cocomponent_fragment**(*root*, *co-comp_ir*

Check whether cocomponent at `cocomp_index` has a cocomponent to its left with same `comp_num`.

INPUT:

- `root` – the forest to which cocomponent belongs

- cocomp_index – index at which cocomponent is present in root

OUTPUT:

`True` if cocomponent at `cocomp_index` has a cocomponent to its left with same `comp_num`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: forest.children[0].comp_num = 1
sage: forest.children[1].comp_num = 1
sage: forest.children[1].children[0].comp_num = 1
sage: forest.children[1].children[1].comp_num = 1
sage: has_left_cocomponent_fragment(forest, 1)
True
sage: has_left_cocomponent_fragment(forest, 0)
False
```

sage.graphs.graph_decompositions.modular_decomposition.**has_right_component_fragment**(*root*, *comp_ina*

Check whether component at `comp_index` has a component to its right with same `comp_num`.

INPUT:

- root – the forest to which component belongs
- comp_index – index at which component is present in root

OUTPUT:

`True` if component at `comp_index` has a component to its right with same `comp_num`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: forest.children[3].comp_num = 1
sage: forest.children[4].comp_num = 1
```

(continues on next page)

(continued from previous page)

```
sage: has_right_component_fragment(forest, 3)
True
sage: has_right_component_fragment(forest, 4)
False
```

sage.graphs.graph_decompositions.modular_decomposition.**has_right_layer_neighbor**(*graph*, *root*, *comp_index*, *vertex_dist*, *vertices_in_compo*

Check whether component at `comp_index` has a connected component to its right with vertices at different level from the source vertex.

INPUT:

- `root` – the forest to which component belongs
- `comp_index` – index at which component is present in root
- `vertex_dist` – dictionary which stores the distance of vertex from source vertex
- `vertices_in_component` – dictionary which stores a list of various vertices in a (co)component

OUTPUT:

`True` if component at `comp_index` has a right layer neighbor, and `False` otherwise.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: vertices_in_component = {}
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
....:     component.index_in_root = index
sage: has_right_layer_neighbor(g, forest, 3, vertex_dist,
```

(continues on next page)

(continued from previous page)

```
....:                             vertices_in_component)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**is_component_connected**(*graph*, *index1*, *index2*, *vertices_in_componen*

Check whether the two specified (co)components are connected.

INPUT:

- `graph` – Graph whose MD tree needs to be computed
- `index1` – index of the first (co)component
- `index2` – index of the second (co)component
- `vertices_in_component` – dictionary which maps index i to list of vertices in the tree at the index i in the forest

OUTPUT:

`True` if the (co)components are connected else `False`

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(6, 1)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertices_in_component = {}
sage: for index, component in enumerate(forest.children):
....:     vertices_in_component[index] = get_vertices(component)
sage: is_component_connected(g, 0, 1, vertices_in_component)
False
sage: is_component_connected(g, 0, 3, vertices_in_component)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**maximal_subtrees_with_leaves_in_x**(*ro*
v,
x,
ve
te
tr
le

Refine the forest based on the active edges(x) of vertex v

INPUT:

- `root` – the forest which needs to be assembled into a MD tree
- `v` – the vertex used to refine
- `x` – set of vertices connected to v and at different distance from source compared to v
- `vertex_status` – dictionary mapping the vertex to the position w.r.t source
- `tree_left_of_source` – flag indicating whether tree is
- `level` – indicates the recursion level, 0 for root

OUTPUT:

`[contained_in_x, split]` where `contained_in_x` is `True` if all vertices in root are subset of x else `False` and `split` is the split which occurred at any node in root

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
```

(continues on next page)

(continued from previous page)

```
sage: x = {u for u in g.neighbor_iterator(2)
....:            if vertex_dist[u] != vertex_dist[2]}
sage: maximal_subtrees_with_leaves_in_x(forest, 2, x, vertex_status,
....:                                   False, 0)
sage: forest
FOREST [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        PARALLEL [NORMAL [6], NORMAL [7]], NORMAL [1]]
sage: x = {u for u in g.neighbor_iterator(1)
....:            if vertex_dist[u] != vertex_dist[1]}
sage: maximal_subtrees_with_leaves_in_x(forest, 1, x, vertex_status,
....:                                   False, 0)
sage: forest
FOREST [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        PARALLEL [PARALLEL [NORMAL [6]], PARALLEL [NORMAL [7]]],
        NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**md_tree_to_graph**(*root*)

Create a graph having the given MD tree.

For the prime nodes we use that every path of length 4 or more is prime.

TODO: accept a function that generates prime graphs as a parameter and use that in the prime nodes.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: tup1 = (NodeType.PRIME, 1, (NodeType.SERIES, 2, 3),
....:         (NodeType.PARALLEL, 4, 5), 6)
sage: tree1 = nested_tuple_to_tree(tup1)
sage: g1 = md_tree_to_graph(tree1)
sage: g2 = Graph({1: [2, 3], 2: [1, 3, 4, 5], 3: [1, 2, 4, 5],
....:             4: [2, 3, 6], 5: [2, 3, 6], 6: [4, 5]})
sage: g1.is_isomorphic(g2)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**modular_decomposition**(*graph*)

Compute the modular decomposition tree of `graph`.

The tree structure is represented in form of nested lists. A tree node is an object of type Node. The Node object further contains a list of its children

INPUT:

- `graph` – the graph for which modular decomposition tree needs to be computed

OUTPUT:

A nested list representing the modular decomposition tree computed for the graph

EXAMPLES:

The Icosahedral graph is Prime:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: print_md_tree(modular_decomposition(graphs.IcosahedralGraph()))
PRIME
      8
      5
      1
      11
```

(continues on next page)

(continued from previous page)

```
      7
      0
      6
      9
      2
      4
      10
      3
```

The Octahedral graph is not Prime:

```
sage: print_md_tree(modular_decomposition(graphs.OctahedralGraph()))
SERIES
      PARALLEL
        2
        3
      PARALLEL
        1
        4
      PARALLEL
        0
        5
```

Tetrahedral Graph is Series:

```
sage: print_md_tree(modular_decomposition(graphs.TetrahedralGraph()))
SERIES
      3
      2
      1
      0
```

Modular Decomposition tree containing both parallel and series modules:

```
sage: d = {2:[4,3,5], 1:[4,3,5], 5:[3,2,1,4], 3:[1,2,5], 4:[1,2,5]}
sage: g = Graph(d)
sage: print_md_tree(modular_decomposition(g))
SERIES
      5
      PARALLEL
        3
        4
      PARALLEL
        1
        2
```

sage.graphs.graph_decompositions.modular_decomposition.**nested_tuple_to_tree**(*nest*)

Turn a tuple representing the modular decomposition into a tree.

INPUT:

- `nest` – a nested tuple of the form returned by *tree_to_nested_tuple()*

OUTPUT:

The root node of a modular decomposition tree.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: tree = (NodeType.SERIES, 1, 2, (NodeType.PARALLEL, 3, 4))
sage: print_md_tree(nested_tuple_to_tree(tree))
SERIES
 1
 2
 PARALLEL
  3
  4
```

sage.graphs.graph_decompositions.modular_decomposition.**number_cocomponents**(*root*, *vertex_status*)

Number the cocomponents to the left of SOURCE vertex in the forest input to the assembly phase

INPUT:

- `root` – the forest which contains the cocomponents and components
- `vertex_status` – dictionary which stores the position of vertex w.r.t SOURCE

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(2, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.LEFT_OF_SOURCE,
....:                  7: VertexPosition.LEFT_OF_SOURCE}
sage: number_cocomponents(forest, vertex_status)
sage: forest.children[1].children[0].comp_num
1
sage: forest.children[1].children[1].comp_num
2
```

sage.graphs.graph_decompositions.modular_decomposition.**number_components**(*root*, *vertex_status*)

Number the components to the right of SOURCE vertex in the forest input to the assembly phase

INPUT:

- `root` – the forest which contains the components and cocomponents
- `vertex_status` – dictionary which stores the position of vertex w.r.t SOURCE

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.append(series_node)
sage: forest.children.append(parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.RIGHT_OF_SOURCE,
....:                  5: VertexPosition.RIGHT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: number_components(forest, vertex_status)
sage: forest.children[-1].children[0].comp_num
2
sage: forest.children[-1].children[1].comp_num
3
```

sage.graphs.graph_decompositions.modular_decomposition.**permute_decomposition**(**args*, ***kwargs*)

Check that a graph and its permuted relabeling have the same modular decomposition.

We generate a `trials` random graphs and then generate an isomorphic graph by relabeling the original graph. We then verify

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: permute_decomposition(30, habib_maurer_algorithm, 10, 0.5)
```

sage.graphs.graph_decompositions.modular_decomposition.**print_md_tree**(*root*)

Print the modular decomposition tree

INPUT:

- `root` – root of the modular decomposition tree

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: print_md_tree(modular_decomposition(graphs.IcosahedralGraph()))
PRIME
      8
      5
      1
      11
      7
      0
      6
      9
      2
      4
```

(continues on next page)

(continued from previous page)

```
    10
    3
```

sage.graphs.graph_decompositions.modular_decomposition.**promote_child**(*root*)

Perform the promotion phase on the forest *root*.

If marked parent has no children it is removed, if it has one child then it is replaced by its child

INPUT:

- `root` – the forest which needs to be promoted

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(4, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: refine(g, forest, vertex_dist, vertex_status)
sage: promote_right(forest)
sage: promote_child(forest)
sage: forest
FOREST [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        NORMAL [7], NORMAL [6], NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**promote_left**(*root*)

Perform the promotion phase on the forest root.

If child and parent both are marked by LEFT_SPLIT then child is removed and placed just before the parent

INPUT:

- `root` – The forest which needs to be promoted

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(4, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:           create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: x = {u for u in g.neighbor_iterator(2)
....:            if vertex_dist[u] != vertex_dist[2]}
sage: maximal_subtrees_with_leaves_in_x(forest, 2, x, vertex_status,
....:                                   False, 0)
sage: promote_left(forest)
sage: forest
FOREST [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
        PARALLEL [NORMAL [6]], PARALLEL [NORMAL [7]],
        PARALLEL [], NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**promote_right**(*root*)

Perform the promotion phase on the forest root.

If child and parent both are marked by RIGHT_SPLIT then child is removed and placed just after the parent

INPUT:

- `root` – the forest which needs to be promoted

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
```

(continues on next page)

(continued from previous page)

```
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(4, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: refine(g, forest, vertex_dist, vertex_status)
sage: promote_right(forest)
sage: forest
FOREST [NORMAL [2], SERIES [SERIES [NORMAL [4]], SERIES [NORMAL [5]]],
        NORMAL [3], PARALLEL [], PARALLEL [NORMAL [7]],
        PARALLEL [NORMAL [6]], NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**random_md_tree**(*max_depth*, *max_fan_out*, *leaf_probability*)

Create a random MD tree.

INPUT:

- `max_depth` – the maximum depth of the tree.
- `max_fan_out` – the maximum number of children a node can have (must be >=4 as a prime node must have at least 4 vertices).
- `leaf_probability` – the probability that a subtree is a leaf

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: tree_to_nested_tuple(random_md_tree(2, 5, 0.5))
(PRIME, [0, 1, (PRIME, [2, 3, 4, 5, 6]), 7, (PARALLEL, [8, 9, 10])])
```

sage.graphs.graph_decompositions.modular_decomposition.**recreate_decomposition**(**args*, ***kwargs*)

Verify that we can recreate a random MD tree.

We create a random MD tree, then create a graph having that decomposition, then find a modular decomposition for that graph, and verify that the two modular decomposition trees are equivalent.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: recreate_decomposition(3, habib_maurer_algorithm, 4, 6, 0.5,
....:                        verbose=False)
```

sage.graphs.graph_decompositions.modular_decomposition.**recursively_number_parts**(*part_root*, *part_num*, *by_type*)

Recursively number the nodes in the (co)components(parts).

If the `node_type` of `part_root` is same as `by_type` then `part_num` is incremented for subtree at each child of `part_root` else part is numbered by `part_num`.

INPUT:

- `part_root` – root of the part to be numbered
- `part_num` – input number to be used as reference for numbering the (co)components
- `by_type` – type which determines how numbering is done

OUTPUT:

The value incremented to `part_num`.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: recursively_number_parts(series_node, 1, NodeType.SERIES)
2
sage: series_node.comp_num
1
sage: series_node.children[0].comp_num
1
sage: series_node.children[1].comp_num
2
```

sage.graphs.graph_decompositions.modular_decomposition.**refine**(*graph*, *root*, *vertex_dist*, *vertex_status*)

Refine the forest based on the active edges

INPUT:

- `graph` – graph whose MD tree needs to be computed
- `root` – the forest which needs to be assembled into a MD tree
- `vertex_dist` – dictionary mapping the vertex with distance from the source
- `vertex_status` – dictionary mapping the vertex to the position w.r.t. source

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = Graph()
sage: g.add_vertices([1, 2, 3, 4, 5, 6, 7])
sage: g.add_edge(2, 3)
sage: g.add_edge(4, 3)
```

(continues on next page)

(continued from previous page)

```
sage: g.add_edge(5, 3)
sage: g.add_edge(2, 6)
sage: g.add_edge(2, 7)
sage: g.add_edge(2, 1)
sage: g.add_edge(6, 1)
sage: g.add_edge(4, 2)
sage: g.add_edge(5, 2)
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
....:                           create_normal_node(7)]
sage: forest.children.insert(1, series_node)
sage: forest.children.insert(3, parallel_node)
sage: vertex_status = {2: VertexPosition.LEFT_OF_SOURCE,
....:                  3: VertexPosition.SOURCE,
....:                  1: VertexPosition.RIGHT_OF_SOURCE,
....:                  4: VertexPosition.LEFT_OF_SOURCE,
....:                  5: VertexPosition.LEFT_OF_SOURCE,
....:                  6: VertexPosition.RIGHT_OF_SOURCE,
....:                  7: VertexPosition.RIGHT_OF_SOURCE}
sage: vertex_dist = {2: 1, 4: 1, 5: 1, 3: 0, 6: 2, 7: 2, 1: 3}
sage: refine(g, forest, vertex_dist, vertex_status)
sage: forest
FOREST [NORMAL [2], SERIES [NORMAL [4], NORMAL [5]], NORMAL [3],
       PARALLEL [PARALLEL [NORMAL [6]], PARALLEL [NORMAL [7]]],
       NORMAL [1]]
```

sage.graphs.graph_decompositions.modular_decomposition.**relabel_tree**(*root*, *perm*)

Relabel the leaves of a tree according to a dictionary

INPUT:

- `root` – the root of the tree
- `perm` – a function, dictionary, list, permutation, or `None` representing the relabeling. See *relabel()* for description of the permutation input.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: tuple_tree = (NodeType.SERIES, 1, 2, (NodeType.PARALLEL, 3, 4))
sage: tree = nested_tuple_to_tree(tuple_tree)
sage: print_md_tree(relabel_tree(tree, (4,3,2,1)))
SERIES
 4
 3
 PARALLEL
  2
  1
```

sage.graphs.graph_decompositions.modular_decomposition.**test_gamma_modules**(**args*, ***kwargs*)

Verify that the vertices of each gamma class of a random graph are modules of that graph.

INPUT:

- `trials` – the number of trials to run
- `vertices` – the size of the graph to use
- `prob` – the probability that any given edge is in the graph. See `RandomGNP()` for more details.
- `verbose` – print information on each trial.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: test_gamma_modules(3, 7, 0.5)
```

sage.graphs.graph_decompositions.modular_decomposition.**test_maximal_modules**(*tree_root*, *graph*)

Test the maximal nature of modules in a modular decomposition tree.

Suppose the module $M = [M_1, M_2, \cdots, n]$ is the input modular decomposition tree. Algorithm forms pairs like $(M_1, M_2), (M_1, M_3), \cdots, (M_1, M_n)$; $(M_2, M_3), (M_2, M_4), \cdots, (M_2, M_n)$; $\cdots$ and so on and tries to form a module using the pair. If the module formed has same type as M and is of type `SERIES` or `PARALLEL` then the formed module is not considered maximal. Otherwise it is considered maximal and M is not a modular decomposition tree.

INPUT:

- `tree_root` – modular decomposition tree whose modules are tested for maximal nature
- `graph` – graph whose modular decomposition tree is tested

OUTPUT:

`True` if all modules at first level in the modular decomposition tree are maximal in nature

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.HexahedralGraph()
sage: test_maximal_modules(modular_decomposition(g), g)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**test_modular_decomposition**(*tree_root*, *graph*)

Test the input modular decomposition tree using recursion.

INPUT:

- `tree_root` – root of the modular decomposition tree to be tested
- `graph` – graph whose modular decomposition tree needs to be tested

OUTPUT:

`True` if input tree is a modular decomposition else `False`

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.HexahedralGraph()
sage: test_modular_decomposition(modular_decomposition(g), g)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**test_module**(*module*, *graph*)

Test whether input module is actually a module

INPUT:

- `module` – module which needs to be tested
- `graph` – input sage graph which contains the module

OUTPUT:

`True` if input module is a module by definition else `False`

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.HexahedralGraph()
sage: tree_root = modular_decomposition(g)
sage: test_module(tree_root, g)
True
sage: test_module(tree_root.children[0], g)
True
```

sage.graphs.graph_decompositions.modular_decomposition.**tree_to_nested_tuple**(*root*)

Convert a modular decomposition tree to a nested tuple.

INPUT:

- `root` – the root of the modular decomposition tree

OUTPUT:

A tuple whose first element is the type of the root of the tree and whose subsequent nodes are either vertex labels in the case of leaves or tuples representing the child subtrees.

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: g = graphs.OctahedralGraph()
sage: tree_to_nested_tuple(modular_decomposition(g))
(SERIES, [(PARALLEL, [2, 3]), (PARALLEL, [1, 4]), (PARALLEL, [0, 5])])
```

sage.graphs.graph_decompositions.modular_decomposition.**update_comp_num**(*node*)

Set the `comp_num` of `node` to the `comp_num` of its first child.

INPUT:

- `node` – node whose comp_num needs to be updated

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.modular_decomposition import *
sage: forest = Node(NodeType.FOREST)
sage: forest.children = [create_normal_node(2),
....:                    create_normal_node(3), create_normal_node(1)]
sage: series_node = Node(NodeType.SERIES)
sage: series_node.comp_num = 2
sage: series_node.children = [create_normal_node(4),
....:                         create_normal_node(5)]
sage: series_node.children[0].comp_num = 3
sage: parallel_node = Node(NodeType.PARALLEL)
sage: parallel_node.children = [create_normal_node(6),
```

(continues on next page)

(continued from previous page)

```
....:                            create_normal_node(7)]
sage: forest.children.insert(0, series_node)
sage: forest.children.insert(3, parallel_node)
sage: update_comp_num(forest)
sage: series_node.comp_num
3
sage: forest.comp_num
2
```

5.25 Decomposition by clique minimal separators

This module implements methods related to the decomposition of a graph by clique minimal separators. See [TY1984] and [BPS2010] for more details on the algorithms.

5.25.1 Methods

sage.graphs.graph_decompositions.clique_separators.**atoms_and_clique_separators**(*G*, *tree=False*, *rooted_tree=Fals* *sep-* *a-* *ra-* *tors=False*)

Return the atoms of the decomposition of G by clique minimal separators.

Let $G = (V, E)$ be a graph. A set $S \subset V$ is a clique separator if $G[S]$ is a clique and the graph $G \setminus S$ has at least 2 connected components. Let $C \subset V$ be the vertices of a connected component of $G \setminus S$. The graph $G[C + S]$ is an *atom* if it has no clique separator.

This method implements the algorithm proposed in [BPS2010], that improves upon the algorithm proposed in [TY1984], for computing the atoms and the clique minimal separators of a graph. This algorithm is based on the *maximum_cardinality_search_M()* graph traversal and has time complexity in $O(|V| \cdot |E|)$.

If the graph is not connected, we insert empty separators between the lists of separators of each connected components. See the examples below for more details.

INPUT:

- `G` – a Sage graph
- `tree` – boolean (default: `False`); whether to return the result as a directed tree in which internal nodes are clique separators and leaves are the atoms of the decomposition. Since a clique separator is repeated when its removal partition the graph into 3 or more connected components, vertices are labels by tuples (i, S), where S is the set of vertices of the atom or the clique separator, and $0 \leq i \leq |T|$.
- `rooted_tree` – boolean (default: `False`); whether to return the result as a `LabelledRootedTree`. When `tree` is `True`, this parameter is ignored.
- `separators` – boolean (default: `False`); whether to also return the complete list of separators considered during the execution of the algorithm. When `tree` or `rooted_tree` is `True`, this parameter is ignored.

OUTPUT:

- By default, return a tuple (A, S_c), where A is the list of atoms of the graph in the order of dicovery, and S_c is the list of clique separators, with possible repetitions, in the order the separator has been considered. If furthermore `separators` is `True`, return a tuple (A, S_h, S_c), where S_c is the list of considered separators of the graph in the order they have been considered.
- When `tree` is `True`, format the result as a directed tree
- When `rooted_tree` is `True` and `tree` is `False`, format the ouput as a `LabelledRootedTree`

EXAMPLES:

Example of [BPS2010]:

```
sage: G = Graph({'a': ['b', 'k'], 'b': ['c'], 'c': ['d', 'j', 'k'],
....:            'd': ['e', 'f', 'j', 'k'], 'e': ['g'],
....:            'f': ['g', 'j', 'k'], 'g': ['j', 'k'], 'h': ['i', 'j'],
....:            'i': ['k'], 'j': ['k']})
sage: atoms, cliques = G.atoms_and_clique_separators()
sage: sorted(sorted(a) for a in atoms)
[['a', 'b', 'c', 'k'],
 ['c', 'd', 'j', 'k'],
 ['d', 'e', 'f', 'g', 'j', 'k'],
 ['h', 'i', 'j', 'k']]
sage: sorted(sorted(c) for c in cliques)
[['c', 'k'], ['d', 'j', 'k'], ['j', 'k']]
sage: T = G.atoms_and_clique_separators(tree=True)
sage: T.is_tree()
True
sage: T.diameter() == len(atoms)
True
sage: all(u[1] in atoms for u in T if T.degree(u) == 1)
True
sage: all(u[1] in cliques for u in T if T.degree(u) != 1)
True
```

A graph without clique separator:

```
sage: G = graphs.CompleteGraph(5)
sage: G.atoms_and_clique_separators()
([{0, 1, 2, 3, 4}], [])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
{0, 1, 2, 3, 4}
```

Graphs with several biconnected components:

```
sage: G = graphs.PathGraph(4)
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _____{2}_____
 /             /
{2, 3}     ___{1}___
          /         /
       {1, 2}    {0, 1}

sage: G = graphs.WindmillGraph(3, 4)
sage: G.atoms_and_clique_separators()
([{0, 1, 2}, {0, 3, 4}, {0, 5, 6}, {0, 8, 7}], [{0}, {0}, {0}])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  __________{0}__________
 /                       /
```

(continues on next page)

(continued from previous page)

```
{0, 1, 2}   ________{0}_______
           /                  /
          {0, 3, 4}    _____{0}___
                      /           /
                     {0, 8, 7} {0, 5, 6}
```

When the removal of a clique separator results in $k > 2$ connected components, this separator is repeated $k - 1$ times, but the repetitions are not necessarily contiguous:

```
sage: G = Graph(2)
sage: for i in range(5):
....:     G.add_cycle([0, 1, G.add_vertex()])
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  __________{0, 1}_____
 /                     /
{0, 1, 4}   _________{0, 1}_____
           /                    /
          {0, 1, 2}   ________{0, 1}___
                     /                 /
                    {0, 1, 3}   _____{0, 1}
                               /           /
                              {0, 1, 5} {0, 1, 6}

sage: G = graphs.StarGraph(3)
sage: G.subdivide_edges(G.edges(), 2)
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{5}_______
 /                 /
{1, 5}   _______{7}_______
        /                  /
       {2, 7}   _______{9}_______
               /                  /
              {9, 3}   _______{6}_______
                      /                  /
                     {6, 7}   _______{4}_____
                             /               /
                            {4, 5}   ______{0}_____
                                    /              /
                                   {0, 6}   _____{8}_____
                                           /             /
                                          {8, 9}   ___{0}___
                                                  /        /
                                                 {0, 8} {0, 4}
```

If the graph is not connected, we insert empty separators between the lists of separators of each connected components. For instance, let G be a graph with 3 connected components. The method returns the list $S_c = [S_0, \cdots, S_i, \ldots, S_j, \ldots, S_{k-1}]$ of k clique separators, where i and j are the indexes of the inserted empty separators and $0 \leq i < j < k - 1$. The method also returns the list $A = [A_0, \ldots, S_k]$ of the $k + 1$ atoms, with $k + 1 \geq 3$. The lists of atoms and clique separators of each of the connected components are respectively $[A_0, \ldots, A_i]$ and $[S_0, \ldots, S_{i-1}]$, $[A_{i+1}, \ldots, A_j]$ and $[S_{i+1}, \ldots, S_{j-1}]$, and $[A_{j+1}, \ldots, A_k]$ and $[S_{j+1}, \ldots, S_{k-1}]$. One can check that for each connected component, we get one atom more than clique separators:

```
sage: G = graphs.PathGraph(3) * 3
sage: A, Sc = G.atoms_and_clique_separators()
sage: A
```

(continues on next page)

(continued from previous page)

```
[{1, 2}, {0, 1}, {4, 5}, {3, 4}, {8, 7}, {6, 7}]
sage: Sc
[{1}, {}, {4}, {}, {7}]
sage: i , j = [i for i, s in enumerate(Sc) if not s]
sage: i, j
(1, 3)
sage: A[:i+1], Sc[:i]
([{1, 2}, {0, 1}], [{1}])
sage: A[i+1:j+1], Sc[i+1:j]
([{4, 5}, {3, 4}], [{4}])
sage: A[j+1:], Sc[j+1:]
([{8, 7}, {6, 7}], [{7}])
sage: I = [-1, i, j, len(Sc)]
sage: for i, j in zip(I[:-1], I[1:]):
....:     print(A[i+1:j+1], Sc[i+1:j])
[{1, 2}, {0, 1}] [{1}]
[{4, 5}, {3, 4}] [{4}]
[{8, 7}, {6, 7}] [{7}]
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{1}_______
 /                 /
{1, 2}    _______{}_______
         /               /
        {0, 1}    ______{4}______
                 /              /
                {4, 5}    ____{}_____
                         /           /
                        {3, 4}    __{7}__
                                 /       /
                                {6, 7} {8, 7}
```

Loops and multiple edges are ignored:

```
sage: G.allow_loops(True)
sage: G.add_edges([(u, u) for u in G])
sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edges())
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  _______{1}_______
 /                 /
{1, 2}    _______{}_______
         /               /
        {0, 1}    ______{4}______
                 /              /
                {4, 5}    ____{}_____
                         /           /
                        {3, 4}    __{7}__
                                 /       /
                                {6, 7} {8, 7}
```

We can check that the returned list of separators is valid:

```
sage: G = graphs.RandomGNP(50, .1)
sage: while not G.is_connected():
....:     G = graphs.RandomGNP(50, .1)
sage: _, separators, _ = G.atoms_and_clique_separators(separators=True)
sage: for S in separators:
```

(continues on next page)

(continued from previous page)

```
....:     H = G.copy()
....:     H.delete_vertices(S)
....:     if H.is_connected():
....:         raise ValueError("something goes wrong")
```

sage.graphs.graph_decompositions.clique_separators.**make_labelled_rooted_tree**(*atoms*, *cliques*)

Return a LabelledRootedTree of atoms and cliques.

The atoms are the leaves of the tree and the cliques are the internal vertices. The number of atoms is the number of cliques plus one.

EXAMPLES:

```
sage: G = graphs.PathGraph(5)
sage: ascii_art(G.atoms_and_clique_separators(rooted_tree=True))
  ______{3}_____
 /              /
{3, 4}    _____{2}____
         /            /
        {2, 3}    __{1}__
                 /       /
                {0, 1} {1, 2}
```

sage.graphs.graph_decompositions.clique_separators.**make_tree**(*atoms*, *cliques*)

Return a tree of atoms and cliques.

The atoms are the leaves of the tree and the cliques are the internal vertices. The number of atoms is the number of cliques plus one.

As a clique may appear several times in the list `cliques`, vertices are numbered by pairs (i, S), where $0 \leq i < |atoms| + |cliques|$ and S is either an atom or a clique.

The root of the tree is the only vertex with even or null degree, i.e., 0 if `cliques` is empty and 2 otherwise. When `cliques` is not empty, other internal vertices (each of which is a clique) have degree 3, and the leaves (vertices of degree 1) are the atoms.

INPUT:

- `atoms` – list of atoms
- `cliques` – list of cliques

EXAMPLES:

```
sage: from sage.graphs.graph_decompositions.clique_separators import make_tree
sage: G = graphs.Grid2dGraph(2, 4)
sage: A, Sc = G.atoms_and_clique_separators()
sage: T = make_tree(A, Sc)
sage: all(u[1] in A for u in T if T.degree(u) == 1)
True
sage: all(u[1] in Sc for u in T if T.degree(u) != 1)
True
```

5.26 Convexity properties of graphs

This class gathers the algorithms related to convexity in a graph. It implements the following methods:

ConvexityProperties.hull()	Return the convex hull of a set of vertices
ConvexityProperties.hull_number()	Compute the hull number of a graph and a corresponding generating set

These methods can be used through the *ConvexityProperties* object returned by *Graph.convexity_properties()*.

AUTHORS:

- Nathann Cohen

5.26.1 Methods

class sage.graphs.convexity_properties.**ConvexityProperties**

Bases: object

This class gathers the algorithms related to convexity in a graph.

Definitions

A set $S \subseteq V(G)$ of vertices is said to be convex if for all $u, v \in S$ the set S contains all the vertices located on a shortest path between u and v. Alternatively, a set S is said to be convex if the distances satisfy $\forall u, v \in S, \forall w \in V \backslash S : d_G(u, w) + d_G(w, v) > d_G(u, v)$.

The convex hull $h(S)$ of a set S of vertices is defined as the smallest convex set containing S.

It is a closure operator, as trivially $S \subseteq h(S)$ and $h(h(S)) = h(S)$.

What this class contains

As operations on convex sets generally involve the computation of distances between vertices, this class' purpose is to cache that information so that computing the convex hulls of several different sets of vertices does not imply recomputing several times the distances between the vertices.

In order to compute the convex hull of a set S it is possible to write the following algorithm:

> For any pair u, v of elements in the set S, and for any vertex w outside of it, add w to S if $d_G(u, w) + d_G(w, v) = d_G(u, v)$. When no vertex can be added anymore, the set S is convex

The distances are not actually that relevant. The same algorithm can be implemented by remembering for each pair u, v of vertices the list of elements w satisfying the condition, and this is precisely what this class remembers, encoded as bitsets to make storage and union operations more efficient.

Note:

- This class is useful if you compute the convex hulls of many sets in the same graph, or if you want to compute the hull number itself as it involves many calls to *hull()*
- Using this class on non-connected graphs is a waste of space and efficiency ! If your graph is disconnected, the best for you is to deal independently with each connected component, whatever you are doing.

Possible improvements

When computing a convex set, all the pairs of elements belonging to the set S are enumerated several times.

- There should be a smart way to avoid enumerating pairs of vertices which have already been tested. The cost of each of them is not very high, so keeping track of those which have been tested already may be too expensive to gain any efficiency.

- The ordering in which they are visited is currently purely lexicographic, while there is a Poset structure to exploit. In particular, when two vertices u, v are far apart and generate a set $h(\{u, v\})$ of vertices, all the pairs of vertices $u', v' \in h(\{u, v\})$ satisfy $h(\{u', v'\}) \subseteq h(\{u, v\})$, and so it is useless to test the pair u', v' when both u and v where present.
- The information cached is for any pair u, v of vertices the list of elements z with $d_G(u, w) + d_G(w, v) = d_G(u, v)$. This is not in general equal to $h(\{u, v\})$!

Nothing says these recommandations will actually lead to any actual improvements. There are just some ideas remembered while writing this code. Trying to optimize may well lead to lost in efficiency on many instances.

EXAMPLES:

```
sage: from sage.graphs.convexity_properties import ConvexityProperties
sage: g = graphs.PetersenGraph()
sage: CP = ConvexityProperties(g)
sage: CP.hull([1, 3])
[1, 2, 3]
sage: CP.hull_number()
3
```

hull (*vertices*)

Return the convex hull of a set of vertices.

INPUT:

- `vertices` – A list of vertices.

EXAMPLES:

```
sage: from sage.graphs.convexity_properties import ConvexityProperties
sage: g = graphs.PetersenGraph()
sage: CP = ConvexityProperties(g)
sage: CP.hull([1, 3])
[1, 2, 3]
```

hull_number (*value_only=True*, *verbose=False*)

Compute the hull number and a corresponding generating set.

The hull number $hn(G)$ of a graph G is the cardinality of a smallest set of vertices S such that $h(S) = V(G)$.

INPUT:

- `value_only` – boolean (default: `True`); whether to return only the hull number (default) or a minimum set whose convex hull is the whole graph
- `verbose` – boolean (default: `False`); whether to display information on the LP

COMPLEXITY:

This problem is NP-Hard [HLT1993], but seems to be of the "nice" kind. Update this comment if you fall on hard instances : $-)$

ALGORITHM:

This is solved by linear programming.

As the function $h(S)$ associating to each set S its convex hull is a closure operator, it is clear that any set S_G of vertices such that $h(S_G) = V(G)$ must satisfy $S_G \not\subseteq C$ for any *proper* convex set $C \subsetneq V(G)$. The

following formulation is hence correct

$$\text{Minimize : } \sum_{v \in G} b_v$$

$$\text{Such that : }$$

$$\forall C \subsetneq V(G) \text{ a proper convex set}$$

$$\sum_{v \in V(G) \setminus C} b_v \geq 1$$

Of course, the number of convex sets – and so the number of constraints – can be huge, and hard to enumerate, so at first an incomplete formulation is solved (it is missing some constraints). If the answer returned by the LP solver is a set S generating the whole graph, then it is optimal and so is returned. Otherwise, the constraint corresponding to the set $h(S)$ can be added to the LP, which makes the answer S infeasible, and another solution computed.

This being said, simply adding the constraint corresponding to $h(S)$ is a bit slow, as these sets can be large (and the corresponding constraint a bit weak). To improve it a bit, before being added, the set $h(S)$ is "greedily enriched" to a set S' with vertices for as long as $h(S') \neq V(G)$. This way, we obtain a set S' with $h(S) \subseteq h(S') \subsetneq V(G)$, and the constraint corresponding to $h(S')$ – which is stronger than the one corresponding to $h(S)$ – is added.

This can actually be seen as a hitting set problem on the complement of convex sets.

EXAMPLES:

The Hull number of Petersen's graph:

```
sage: from sage.graphs.convexity_properties import ConvexityProperties
sage: g = graphs.PetersenGraph()
sage: CP = ConvexityProperties(g)
sage: CP.hull_number()
3
sage: generating_set = CP.hull_number(value_only=False)
sage: CP.hull(generating_set)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.27 Weakly chordal graphs

This module deals with everything related to weakly chordal graphs. It currently contains the following functions:

is_long_hole_free()	Tests whether g contains an induced cycle of length at least 5.
is_long_antihole_free()	Tests whether g contains an induced anticycle of length at least 5.
is_weakly_chordal()	Tests whether g is weakly chordal.

Author:

- Birk Eisermann (initial implementation)
- Nathann Cohen (some doc and optimization)
- David Coudert (remove recursion)

5.27.1 Methods

sage.graphs.weakly_chordal.**is_long_antihole_free**(*g*, *certificate=False*)

Tests whether the given graph contains an induced subgraph that is isomorphic to the complement of a cycle of length at least 5.

INPUT:

- `certificate` – boolean (default: `False`)

  Whether to return a certificate. When `certificate = True`, then the function returns

  - `(False, Antihole)` if `g` contains an induced complement of a cycle of length at least 5 returned as `Antihole`.
  - `(True, [])` if `g` does not contain an induced complement of a cycle of length at least 5. For this case it is not known how to provide a certificate.

  When `certificate = False`, the function returns just `True` or `False` accordingly.

ALGORITHM:

This algorithm tries to find a cycle in the graph of all induced $\overline{P_4}$ of g, where two copies $\overline{P}$ and $\overline{P'}$ of $\overline{P_4}$ are adjacent if there exists a (not necessarily induced) copy of $\overline{P_5} = u_1u_2u_3u_4u_5$ such that $\overline{P} = u_1u_2u_3u_4$ and $\overline{P'} = u_2u_3u_4u_5$.

This is done through a depth-first-search. For efficiency, the auxiliary graph is constructed on-the-fly and never stored in memory.

The run time of this algorithm is $O(m^2)$ [NP2007] (where m is the number of edges of the graph).

EXAMPLES:

The Petersen Graph contains an antihole:

```
sage: g = graphs.PetersenGraph()
sage: g.is_long_antihole_free()
False
```

The complement of a cycle is an antihole:

```
sage: g = graphs.CycleGraph(6).complement()
sage: r,a = g.is_long_antihole_free(certificate=True)
sage: r
False
sage: a.complement().is_isomorphic(graphs.CycleGraph(6))
True
```

sage.graphs.weakly_chordal.**is_long_hole_free**(*g*, *certificate=False*)

Tests whether `g` contains an induced cycle of length at least 5.

INPUT:

- `certificate` – boolean (default: `False`)

  Whether to return a certificate. When `certificate = True`, then the function returns

  - `(True, [])` if `g` does not contain such a cycle. For this case, it is not known how to provide a certificate.
  - `(False, Hole)` if `g` contains an induced cycle of length at least 5. `Hole` returns this cycle.

  If `certificate = False`, the function returns just `True` or `False` accordingly.

ALGORITHM:

This algorithm tries to find a cycle in the graph of all induced P_4 of g, where two copies P and P' of P_4 are adjacent if there exists a (not necessarily induced) copy of $P_5 = u_1u_2u_3u_4u_5$ such that $P = u_1u_2u_3u_4$ and $P' = u_2u_3u_4u_5$.

This is done through a depth-first-search. For efficiency, the auxiliary graph is constructed on-the-fly and never stored in memory.

The run time of this algorithm is $O(m^2)$ [NP2007] (where m is the number of edges of the graph) .

EXAMPLES:

The Petersen Graph contains a hole:

```
sage: g = graphs.PetersenGraph()
sage: g.is_long_hole_free()
False
```

The following graph contains a hole, which we want to display:

```
sage: g = graphs.FlowerSnark()
sage: r,h = g.is_long_hole_free(certificate=True)
sage: r
False
sage: Graph(h).is_isomorphic(graphs.CycleGraph(h.order()))
True
```

sage.graphs.weakly_chordal.**is_weakly_chordal**(*g*, *certificate=False*)

Tests whether the given graph is weakly chordal, i.e., the graph and its complement have no induced cycle of length at least 5.

INPUT:

- `certificate` – Boolean value (default: `False`) whether to return a certificate. If `certificate = False`, return `True` or `False` according to the graph. If `certificate = True`, return
  - `(False, forbidden_subgraph)` when the graph contains a forbidden subgraph H, this graph is returned.
  - **`(True, [])` when the graph is weakly chordal.** For this case, it is not known how to provide a certificate.

ALGORITHM:

This algorithm checks whether the graph `g` or its complement contain an induced cycle of length at least 5.

Using is_long_hole_free() and is_long_antihole_free() yields a run time of $O(m^2)$ (where m is the number of edges of the graph).

EXAMPLES:

The Petersen Graph is not weakly chordal and contains a hole:

```
sage: g = graphs.PetersenGraph()
sage: r,s = g.is_weakly_chordal(certificate=True)
sage: r
False
sage: l = s.order()
sage: s.is_isomorphic(graphs.CycleGraph(l))
True
```

5.28 Distances/shortest paths between all pairs of vertices

This module implements a few functions that deal with the computation of distances or shortest paths between all pairs of vertices.

Efficiency : Because these functions involve listing many times the (out)-neighborhoods of (di)-graphs, it is useful in terms of efficiency to build a temporary copy of the graph in a data structure that makes it easy to compute quickly. These functions also work on large volume of data, typically dense matrices of size n^2, and are expected to return corresponding dictionaries of size n^2, where the integers corresponding to the vertices have first been converted to the vertices' labels. Sadly, this last translating operation turns out to be the most time-consuming, and for this reason it is also nice to have a Cython module, and version of these functions that return C arrays, in order to avoid these operations when they are not necessary.

Memory cost : The methods implemented in the current module sometimes need large amounts of memory to return their result. Storing the distances between all pairs of vertices in a graph on 1500 vertices as a dictionary of dictionaries takes around 200MB, while storing the same information as a C array requires 4MB.

5.28.1 The module's main function

The C function `all_pairs_shortest_path_BFS` actually does all the computations, and all the others (except for `Floyd_Warshall`) are just wrapping it. This function begins with copying the graph in a data structure that makes it fast to query the out-neighbors of a vertex, then starts one Breadth First Search per vertex of the (di)graph.

What can this function compute ?

- The matrix of predecessors.

  This matrix P has size n^2, and is such that vertex $P[u, v]$ is a predecessor of v on a shortest uv-path. Hence, this matrix efficiently encodes the information of a shortest uv-path for any $u, v \in G$: indeed, to go from u to v you should first find a shortest $uP[u, v]$-path, then jump from $P[u, v]$ to v as it is one of its outneighbors. Apply recursively and find out what the whole path is !.

- The matrix of distances.

  This matrix has size n^2 and associates to any uv the distance from u to v.

- The vector of eccentricities.

  This vector of size n encodes for each vertex v the distance to vertex which is furthest from v in the graph. In particular, the diameter of the graph is the maximum of these values.

What does it take as input ?

- `gg` a (Di)Graph.
- `unsigned short * predecessors` – a pointer toward an array of size $n^2 \cdot$ sizeof(unsigned short). Set to `NULL` if you do not want to compute the predecessors.
- `unsigned short * distances` – a pointer toward an array of size $n^2 \cdot$ sizeof(unsigned short). The computation of the distances is necessary for the algorithm, so this value can **not** be set to `NULL`.
- `int * eccentricity` – a pointer toward an array of size $n \cdot$ sizeof(int). Set to `NULL` if you do not want to compute the eccentricity.

Technical details

- The vertices are encoded as $1, ..., n$ as they appear in the ordering of `G.vertices()`, unless another ordering is specified by the user.

- Because this function works on matrices whose size is quadratic compared to the number of vertices when computing all distances or predecessors, it uses short variables to store the vertices' names instead of long ones to divide by 2 the size in memory. This means that only the diameter/eccentricities can be computed on a graph of more than 65536 nodes. For information, the current version of the algorithm on a graph with $65536 = 2^{16}$ nodes creates in memory 2 tables on 2^{32} short elements (2bytes each), for a total of 2^{33} bytes or 8 gigabytes. In order to support larger sizes, we would have to replace shorts by 32-bits int or 64-bits int, which would then require respectively 16GB or 32GB.
- In the C version of these functions, infinite distances are represented with `<unsigned short> -1 = 65535` for `unsigned short` variables, and by `INT32_MAX` otherwise. These case happens when the input is a disconnected graph, or a non-strongly-connected digraph.
- A memory error is raised when data structures allocation failed. This could happen with large graphs on computers with low memory space.

Warning: The function `all_pairs_shortest_path_BFS` has **no reason** to be called by the user, even though he would be writing his code in Cython and look for efficiency. This module contains wrappers for this function that feed it with the good parameters. As the function is inlined, using those wrappers actually saves time as it should avoid testing the parameters again and again in the main function's body.

AUTHOR:

- Nathann Cohen (2011)
- David Coudert (2014) – 2sweep, multi-sweep and iFUB for diameter computation

5.28.2 Functions

`sage.graphs.distances_all_pairs.`**`diameter`**(*G*, *algorithm=None*, *source=None*)

Return the diameter of G.

This algorithm returns Infinity if the (di)graph is not connected. It can also quickly return a lower bound on the diameter using the `2sweep`, `2Dsweep` and `multi-sweep` schemes.

INPUT:

- `algorithm` – string (default: `None`); specifies the algorithm to use among:
  - `'standard'` – Computes the diameter of the input (di)graph as the largest eccentricity of its vertices. This is the classical algorithm with time complexity in $O(nm)$.
  - `'2sweep'` – Computes a lower bound on the diameter of an unweighted undirected graph using 2 BFS, as proposed in [MLH2008]. It first selects a vertex v that is at largest distance from an initial vertex source using BFS. Then it performs a second BFS from v. The largest distance from v is returned as a lower bound on the diameter of G. The time complexity of this algorithm is linear in the size of G.
  - `'2Dsweep'` – Computes lower bound on the diameter of an unweighted directed graph using directed version of `2sweep` as proposed in [Broder2000].
  - `'multi-sweep'` – Computes a lower bound on the diameter of an unweighted undirected graph using several iterations of the `2sweep` algorithms [CGHLM2013]. Roughly, it first uses `2sweep` to identify two vertices u and v that are far apart. Then it selects a vertex w that is at same distance from u and v. This vertex w will serve as the new source for another iteration of the `2sweep` algorithm that may improve the current lower bound on the diameter. This process is repeated as long as the lower bound on the diameter is improved.

- `'iFUB'` – The iFUB (iterative Fringe Upper Bound) algorithm, proposed in [CGILM2010], computes the exact value of the diameter of an unweighted undirected graph. It is based on the following observation:

  The diameter of the graph is equal to the maximum eccentricity of a vertex. Let v be any vertex, and let V be partitionned into $A \cup B$ where:

  $$d(v,a) \leq i, \forall a \in A$$
  $$d(v,b) \geq i, \forall b \in B$$

  As all vertices from A are at distance $\leq 2i$ from each other, a vertex $a \in A$ with eccentricity $ecc(a) > 2i$ is at distance $ecc(a)$ from some vertex $b \in B$.

  Consequently, if we have already computed the maximum eccentricity m of all vertices in B and if $m > 2i$, then we do not need to compute the eccentricity of the vertices in A.

  Starting from a vertex v obtained through a multi-sweep computation (which refines the 4sweep algorithm used in [CGHLM2013]), we compute the diameter by computing the eccentricity of all vertices sorted decreasingly according to their distance to v, and stop as allowed by the remark above. The worst case time complexity of the iFUB algorithm is $O(nm)$, but it can be very fast in practice.

- `source` – (default: `None`) vertex from which to start the first BFS. If `source==None`, an arbitrary vertex of the graph is chosen. Raise an error if the initial vertex is not in G. This parameter is not used when `algorithm=='standard'`.

EXAMPLES:

```
sage: from sage.graphs.distances_all_pairs import diameter
sage: G = graphs.PetersenGraph()
sage: diameter(G, algorithm='iFUB')
2
sage: G = Graph({0: [], 1: [], 2: [1]})
sage: diameter(G, algorithm='iFUB')
+Infinity
sage: G = digraphs.Circuit(6)
sage: diameter(G, algorithm='2Dsweep')
5
```

Although max( ) is usually defined as -Infinity, since the diameter will never be negative, we define it to be zero:

```
sage: G = graphs.EmptyGraph()
sage: diameter(G, algorithm='iFUB')
0
```

Comparison of exact algorithms:

```
sage: G = graphs.RandomBarabasiAlbert(100, 2)
sage: d1 = diameter(G, algorithm='standard')
sage: d2 = diameter(G, algorithm='iFUB')
sage: d3 = diameter(G, algorithm='iFUB', source=G.random_vertex())
sage: if d1 != d2 or d1 != d3: print("Something goes wrong!")
```

Comparison of lower bound algorithms:

```
sage: lb2 = diameter(G, algorithm='2sweep')
sage: lbm = diameter(G, algorithm='multi-sweep')
sage: if not (lb2 <= lbm and lbm <= d3): print("Something goes wrong!")
```

`sage.graphs.distances_all_pairs.`**`distances_all_pairs`**`(G)`

Return the matrix of distances in G.

This function returns a double dictionary `D` of vertices, in which the distance between vertices `u` and `v` is `D[u][v]`.

EXAMPLES:

```
sage: from sage.graphs.distances_all_pairs import distances_all_pairs
sage: g = graphs.PetersenGraph()
sage: distances_all_pairs(g)
{0: {0: 0, 1: 1, 2: 2, 3: 2, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2, 9: 2},
1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 2, 5: 2, 6: 1, 7: 2, 8: 2, 9: 2},
2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 2, 5: 2, 6: 2, 7: 1, 8: 2, 9: 2},
3: {0: 2, 1: 2, 2: 1, 3: 0, 4: 1, 5: 2, 6: 2, 7: 2, 8: 1, 9: 2},
4: {0: 1, 1: 2, 2: 2, 3: 1, 4: 0, 5: 2, 6: 2, 7: 2, 8: 2, 9: 1},
5: {0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 0, 6: 2, 7: 1, 8: 1, 9: 2},
6: {0: 2, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2, 6: 0, 7: 2, 8: 1, 9: 1},
7: {0: 2, 1: 2, 2: 1, 3: 2, 4: 2, 5: 1, 6: 2, 7: 0, 8: 2, 9: 1},
8: {0: 2, 1: 2, 2: 2, 3: 1, 4: 2, 5: 1, 6: 1, 7: 2, 8: 0, 9: 2},
9: {0: 2, 1: 2, 2: 2, 3: 2, 4: 1, 5: 2, 6: 1, 7: 1, 8: 2, 9: 0}}
```

sage.graphs.distances_all_pairs.**distances_and_predecessors_all_pairs**(G)

Return the matrix of distances in G and the matrix of predecessors.

Distances : the matrix M returned is of length n^2, and the distance between vertices u and v is $M[u, v]$. The integer corresponding to a vertex is its index in the list `G.vertices()`.

Predecessors : the matrix P returned has size n^2, and is such that vertex $P[u, v]$ is a predecessor of v on a shortest uv-path. Hence, this matrix efficiently encodes the information of a shortest uv-path for any $u, v \in G$: indeed, to go from u to v you should first find a shortest $uP[u, v]$-path, then jump from $P[u, v]$ to v as it is one of its outneighbors.

EXAMPLES:

```
sage: from sage.graphs.distances_all_pairs import distances_and_predecessors_all_
↪pairs
sage: g = graphs.PetersenGraph()
sage: distances_and_predecessors_all_pairs(g)
({0: {0: 0, 1: 1, 2: 2, 3: 2, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2, 9: 2},
  1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 2, 5: 2, 6: 1, 7: 2, 8: 2, 9: 2},
  2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 2, 5: 2, 6: 2, 7: 1, 8: 2, 9: 2},
  3: {0: 2, 1: 2, 2: 1, 3: 0, 4: 1, 5: 2, 6: 2, 7: 2, 8: 1, 9: 2},
  4: {0: 1, 1: 2, 2: 2, 3: 1, 4: 0, 5: 2, 6: 2, 7: 2, 8: 2, 9: 1},
  5: {0: 1, 1: 2, 2: 2, 3: 2, 4: 2, 5: 0, 6: 2, 7: 1, 8: 1, 9: 2},
  6: {0: 2, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2, 6: 0, 7: 2, 8: 1, 9: 1},
  7: {0: 2, 1: 2, 2: 1, 3: 2, 4: 2, 5: 1, 6: 2, 7: 0, 8: 2, 9: 1},
  8: {0: 2, 1: 2, 2: 2, 3: 1, 4: 2, 5: 1, 6: 1, 7: 2, 8: 0, 9: 2},
  9: {0: 2, 1: 2, 2: 2, 3: 2, 4: 1, 5: 2, 6: 1, 7: 1, 8: 2, 9: 0}},
 {0: {0: None, 1: 0, 2: 1, 3: 4, 4: 0, 5: 0, 6: 1, 7: 5, 8: 5, 9: 4},
  1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0, 5: 0, 6: 1, 7: 2, 8: 6, 9: 6},
  2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3, 5: 7, 6: 1, 7: 2, 8: 3, 9: 7},
  3: {0: 4, 1: 2, 2: 3, 3: None, 4: 3, 5: 8, 6: 8, 7: 2, 8: 3, 9: 4},
  4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None, 5: 0, 6: 9, 7: 9, 8: 3, 9: 4},
  5: {0: 5, 1: 0, 2: 7, 3: 8, 4: 0, 5: None, 6: 8, 7: 5, 8: 5, 9: 7},
  6: {0: 1, 1: 6, 2: 1, 3: 8, 4: 9, 5: 8, 6: None, 7: 9, 8: 6, 9: 6},
  7: {0: 5, 1: 2, 2: 7, 3: 2, 4: 9, 5: 7, 6: 9, 7: None, 8: 5, 9: 7},
  8: {0: 5, 1: 6, 2: 3, 3: 8, 4: 3, 5: 8, 6: 8, 7: 5, 8: None, 9: 6},
  9: {0: 4, 1: 6, 2: 7, 3: 4, 4: 9, 5: 7, 6: 9, 7: 9, 8: 6, 9: None}})
```

sage.graphs.distances_all_pairs.**distances_distribution**(G)

Return the distances distribution of the (di)graph in a dictionary.

This method *ignores all edge labels*, so that the distance considered is the topological distance.

OUTPUT:

A dictionary `d` such that the number of pairs of vertices at distance `k` (if any) is equal to $d[k] \cdot |V(G)| \cdot (|V(G)| - 1)$.

Note: We consider that two vertices that do not belong to the same connected component are at infinite distance, and we do not take the trivial pairs of vertices (v, v) at distance 0 into account. Empty (di)graphs and (di)graphs of order 1 have no paths and so we return the empty dictionary `{}`.

EXAMPLES:

An empty Graph:

```
sage: g = Graph()
sage: g.distances_distribution()
{}
```

A Graph of order 1:

```
sage: g = Graph()
sage: g.add_vertex(1)
sage: g.distances_distribution()
{}
```

A Graph of order 2 without edge:

```
sage: g = Graph()
sage: g.add_vertices([1,2])
sage: g.distances_distribution()
{+Infinity: 1}
```

The Petersen Graph:

```
sage: g = graphs.PetersenGraph()
sage: g.distances_distribution()
{1: 1/3, 2: 2/3}
```

A graph with multiple disconnected components:

```
sage: g = graphs.PetersenGraph()
sage: g.add_edge('good','wine')
sage: g.distances_distribution()
{1: 8/33, 2: 5/11, +Infinity: 10/33}
```

The de Bruijn digraph dB(2,3):

```
sage: D = digraphs.DeBruijn(2,3)
sage: D.distances_distribution()
{1: 1/4, 2: 11/28, 3: 5/14}
```

`sage.graphs.distances_all_pairs.`**`eccentricity`**(*G*, *algorithm='standard'*, *vertex_list=None*)

Return the vector of eccentricities in G.

The array returned is of length n, and its i-th component is the eccentricity of the ith vertex in `G.vertices()`.

INPUT:

- G – a Graph or a DiGraph.
- algorithm – string (default: 'standard'); name of the method used to compute the eccentricity of the vertices. Available algorithms are 'standard' which performs a BFS from each vertex and 'bounds' which uses the fast algorithm proposed in [TK2013] for undirected graphs.
- vertex_list – list (default: None); a list of n vertices specifying a mapping from $(0, \ldots, n-1)$ to vertex labels in G. When set, ecc[i] is the eccentricity of vertex vertex_list[i]. When vertex_list is None, ecc[i] is the eccentricity of vertex G.vertices()[i].

EXAMPLES:

```
sage: from sage.graphs.distances_all_pairs import eccentricity
sage: g = graphs.PetersenGraph()
sage: eccentricity(g)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
sage: g.add_edge(0, g.add_vertex())
sage: V = list(g)
sage: eccentricity(g, vertex_list=V)
[2, 2, 3, 3, 2, 2, 3, 3, 3, 3]
sage: eccentricity(g, vertex_list=V[::-1])
[3, 3, 3, 3, 3, 2, 2, 3, 3, 2, 2]
```

sage.graphs.distances_all_pairs.**floyd_warshall**(*gg*, *paths=True*, *distances=False*)

Compute the shortest path/distances between all pairs of vertices.

For more information on the Floyd-Warshall algorithm, see the Wikipedia article Floyd-Warshall_algorithm.

INPUT:

- gg – the graph on which to work.
- paths – boolean (default: True); whether to return the dictionary of shortest paths
- distances – boolean (default: False); whether to return the dictionary of distances

OUTPUT:

Depending on the input, this function return the dictionary of paths, the dictionary of distances, or a pair of dictionaries (distances, paths) where distance[u][v] denotes the distance of a shortest path from u to v and paths[u][v] denotes an inneighbor w of v such that $dist(u, v) = 1 + dist(u, w)$.

Warning: Because this function works on matrices whose size is quadratic compared to the number of vertices, it uses short variables instead of long ones to divide by 2 the size in memory. This means that the current implementation does not run on a graph of more than 65536 nodes (this can be easily changed if necessary, but would require much more memory. It may be worth writing two versions). For information, the current version of the algorithm on a graph with $65536 = 2^{16}$ nodes creates in memory 2 tables on 2^{32} short elements (2bytes each), for a total of 2^{34} bytes or 16 gigabytes. Let us also remember that if the memory size is quadratic, the algorithm runs in cubic time.

Note: When paths = False the algorithm saves roughly half of the memory as it does not have to maintain the matrix of predecessors. However, setting distances=False produces no such effect as the algorithm can not run without computing them. They will not be returned, but they will be stored while the method is running.

EXAMPLES:

Shortest paths in a small grid

```
sage: g = graphs.Grid2dGraph(2,2)
sage: from sage.graphs.distances_all_pairs import floyd_warshall
sage: print(floyd_warshall(g))  # py2
{(0, 1): {(0, 1): None, (1, 0): (0, 0), (0, 0): (0, 1), (1, 1): (0, 1)},
 (1, 0): {(0, 1): (0, 0), (1, 0): None, (0, 0): (1, 0), (1, 1): (1, 0)},
 (0, 0): {(0, 1): (0, 0), (1, 0): (0, 0), (0, 0): None, (1, 1): (0, 1)},
 (1, 1): {(0, 1): (1, 1), (1, 0): (1, 1), (0, 0): (0, 1), (1, 1): None}}
sage: print(floyd_warshall(g))  # py3
{(0, 0): {(0, 0): None, (0, 1): (0, 0), (1, 0): (0, 0), (1, 1): (0, 1)},
 (0, 1): {(0, 1): None, (0, 0): (0, 1), (1, 0): (0, 0), (1, 1): (0, 1)},
 (1, 0): {(1, 0): None, (0, 0): (1, 0), (0, 1): (0, 0), (1, 1): (1, 0)},
 (1, 1): {(1, 1): None, (0, 0): (0, 1), (0, 1): (1, 1), (1, 0): (1, 1)}}
```

Checking the distances are correct

```
sage: g = graphs.Grid2dGraph(5,5)
sage: dist,path = floyd_warshall(g, distances=True)
sage: all(dist[u][v] == g.distance(u, v) for u in g for v in g)
True
```

Checking a random path is valid

```
sage: u,v = g.random_vertex(), g.random_vertex()
sage: p = [v]
sage: while p[0] is not None:
....:   p.insert(0,path[u][p[0]])
sage: len(p) == dist[u][v] + 2
True
```

Distances for all pairs of vertices in a diamond:

```
sage: g = graphs.DiamondGraph()
sage: floyd_warshall(g, paths=False, distances=True)
{0: {0: 0, 1: 1, 2: 1, 3: 2},
 1: {0: 1, 1: 0, 2: 1, 3: 1},
 2: {0: 1, 1: 1, 2: 0, 3: 1},
 3: {0: 2, 1: 1, 2: 1, 3: 0}}
```

sage.graphs.distances_all_pairs.**is_distance_regular**(*G*, *parameters=False*)

Test if the graph is distance-regular

A graph G is distance-regular if for any integers j, k the value of $|\{x : d_G(x,u) = j, x \in V(G)\} \cap \{y : d_G(y,v) = j, y \in V(G)\}|$ is constant for any two vertices $u, v \in V(G)$ at distance i from each other. In particular G is regular, of degree b_0 (see below), as one can take $u = v$.

Equivalently a graph is distance-regular if there exist integers b_i, c_i such that for any two vertices u, v at distance i we have

- $b_i = |\{x : d_G(x,u) = i+1, x \in V(G)\} \cap N_G(v)\}|,\ 0 \leq i \leq d-1$
- $c_i = |\{x : d_G(x,u) = i-1, x \in V(G)\} \cap N_G(v)\}|,\ 1 \leq i \leq d,$

where d is the diameter of the graph. For more information on distance-regular graphs, see the Wikipedia article Distance-regular_graph.

INPUT:

- `parameters` – boolean (default: `False`); if set to `True`, the function returns the pair `(b, c)` of lists of integers instead of a boolean answer (see the definition above)

See also:

- *is_regular()*
- *is_strongly_regular()*

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.is_distance_regular()
True
sage: g.is_distance_regular(parameters = True)
([3, 2, None], [None, 1, 1])
```

Cube graphs, which are not strongly regular, are a bit more interesting:

```
sage: graphs.CubeGraph(4).is_distance_regular()
True
sage: graphs.OddGraph(5).is_distance_regular()
True
```

Disconnected graph:

```
sage: (2*graphs.CubeGraph(4)).is_distance_regular()
True
```

sage.graphs.distances_all_pairs.**shortest_path_all_pairs**(*G*)

Return the matrix of predecessors in G.

The matrix P returned has size n^2, and is such that vertex $P[u, v]$ is a predecessor of v on a shortest uv-path. Hence, this matrix efficiently encodes the information of a shortest uv-path for any $u, v \in G$: indeed, to go from u to v you should first find a shortest $uP[u, v]$-path, then jump from $P[u, v]$ to v as it is one of its outneighbors.

EXAMPLES:

```
sage: from sage.graphs.distances_all_pairs import shortest_path_all_pairs
sage: g = graphs.PetersenGraph()
sage: shortest_path_all_pairs(g)
{0: {0: None, 1: 0, 2: 1, 3: 4, 4: 0, 5: 0, 6: 1, 7: 5, 8: 5, 9: 4},
 1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0, 5: 0, 6: 1, 7: 2, 8: 6, 9: 6},
 2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3, 5: 7, 6: 1, 7: 2, 8: 3, 9: 7},
 3: {0: 4, 1: 2, 2: 3, 3: None, 4: 3, 5: 8, 6: 8, 7: 2, 8: 3, 9: 4},
 4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None, 5: 0, 6: 9, 7: 9, 8: 3, 9: 4},
 5: {0: 5, 1: 0, 2: 7, 3: 8, 4: 0, 5: None, 6: 8, 7: 5, 8: 5, 9: 7},
 6: {0: 1, 1: 6, 2: 1, 3: 8, 4: 9, 5: 8, 6: None, 7: 9, 8: 6, 9: 6},
 7: {0: 5, 1: 2, 2: 7, 3: 2, 4: 9, 5: 7, 6: 9, 7: None, 8: 5, 9: 7},
 8: {0: 5, 1: 6, 2: 3, 3: 8, 4: 3, 5: 8, 6: 8, 7: 5, 8: None, 9: 6},
 9: {0: 4, 1: 6, 2: 7, 3: 4, 4: 9, 5: 7, 6: 9, 7: 9, 8: 6, 9: None}}
```

sage.graphs.distances_all_pairs.**wiener_index**(*G*)

Return the Wiener index of the graph.

The Wiener index of a graph G can be defined in two equivalent ways [KRG1996] :

- $W(G) = \frac{1}{2}\sum_{u,v\in G} d(u, v)$ where $d(u, v)$ denotes the distance between vertices u and v.
- Let Ω be a set of $\frac{n(n-1)}{2}$ paths in G such that Ω contains exactly one shortest $u - v$ path for each set $\{u, v\}$ of vertices in G. Besides, $\forall e \in E(G)$, let $\Omega(e)$ denote the paths from Ω containing e. We then have $W(G) = \sum_{e\in E(G)} |\Omega(e)|$.

EXAMPLES:

From [GYLL1993], cited in [KRG1996]:

```
sage: g=graphs.PathGraph(10)
sage: w=lambda x: (x*(x*x -1)/6)
sage: g.wiener_index()==w(10)
True
```

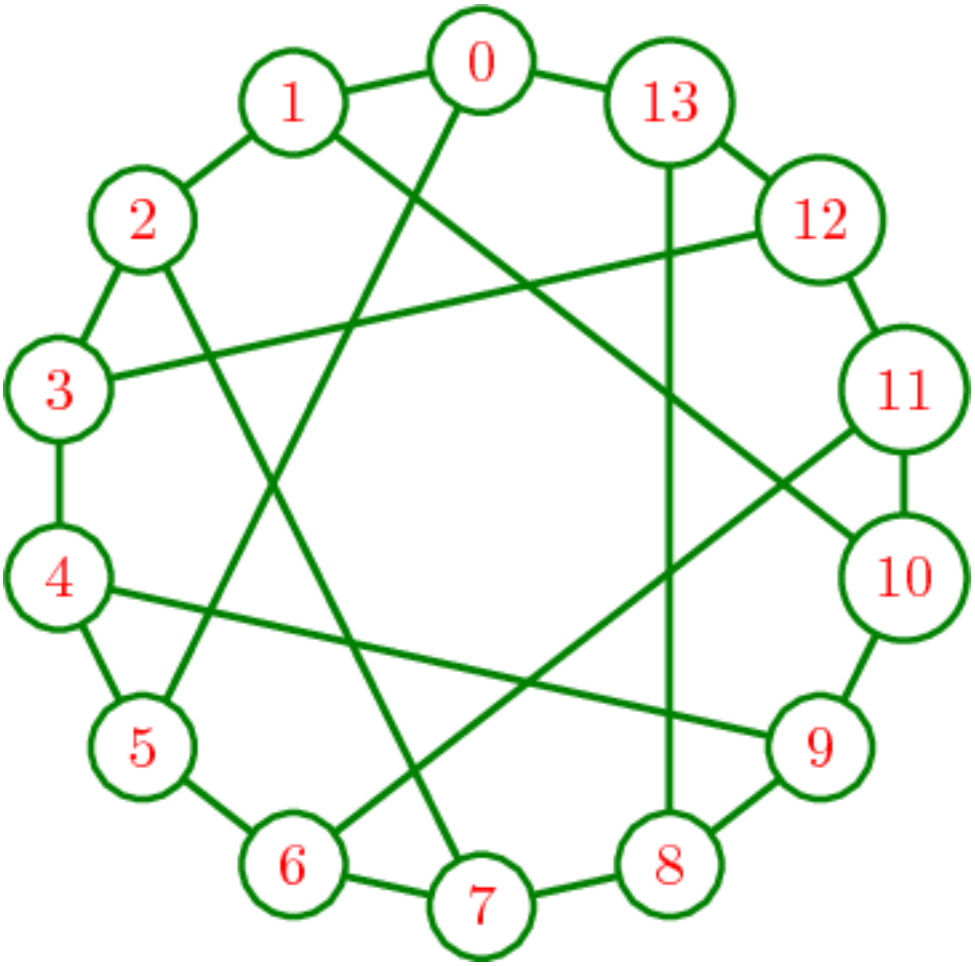
5.29 LaTeX options for graphs

This module provides a class to hold, manipulate and employ various options for rendering a graph in LaTeX, in addition to providing the code that actually generates a LaTeX representation of a (combinatorial) graph.

AUTHORS:

- Rob Beezer (2009-05-20): *GraphLatex* class
- Fidel Barerra Cruz (2009-05-20): `tkz-graph` commands to render a graph
- Nicolas M. Thiéry (2010-02): dot2tex/graphviz interface
- Rob Beezer (2010-05-29): Extended range of `tkz-graph` options

5.29.1 LaTeX Versions of Graphs



Many mathematical objects in Sage have LaTeX representations, and graphs are no exception. For a graph `g`, the command `view(g)`, issued at the Sage command line or in the notebook, will create a graphic version of `g`. Similarly, `latex(g)` will return a (long) string that is a representation of the graph in LaTeX. Other ways of employing LaTeX in Sage, such as `%latex` in a notebook cell, or the Typeset checkbox in the notebook, will handle `g` appropriately.

Support through the `tkz-graph` package is by Alain Matthes, the author of `tkz-graph`, whose work can be found at his Altermundus.com site.

The range of possible options for customizing the appearance of a graph are carefully documented at *sage.graphs.graph_latex.GraphLatex.set_option()*. As a broad overview, the following options are supported:

- Pre-built Styles: the pre-built styles of the tkz-graph package provide nice drawings quickly
- Dimensions: can be specified in natural units, then uniformly scaled after design work
- Vertex Colors: the perimeter and fill color for vertices can be specified, including on a per-vertex basis
- Vertex Shapes: may be circles, shaded spheres, rectangles or diamonds, including on a per-vertex basis
- Vertex Sizes: may be specified as minimums, and will automatically sized to contain vertex labels, including on a per-vertex basis
- Vertex Labels: can use latex formatting, and may have their colors specified, including on a per-vertex basis
- Vertex Label Placement: can be interior to the vertex, or external at a configurable location

- Edge Colors: a solid color with or without a second color down the middle, on a per-edge basis
- Edge Thickness: can be set, including on a per-edge basis
- Edge Labels: can use latex formatting, and may have their colors specified, including on a per-edge basis
- Edge Label Placement: can be to the left, right, above, below, inline, and then sloped or horizontal
- Digraph Edges: are slightly curved, with arrowheads
- Loops: may be specified by their size, and with a direction equaling one of the four compass points

To use LaTeX in Sage you of course need a working TeX installation and it will work best if you have the `dvipng` and `convert` utilities. For graphs you need the `tkz-graph.sty` and `tkz-berge.sty` style files of the tkz-graph package. TeX, dvipng, and convert should be widely available through package managers or installers. You may need to install the tkz-graph style files in the appropriate locations, a task beyond the scope of this introduction. Primary locations for these programs are:

- TeX: http://ctan.org/
- dvipng: http://sourceforge.net/projects/dvipng/
- convert: http://www.imagemagick.org (the ImageMagick suite)
- tkz-graph: http://altermundus.com/pages/tkz/

Customizing the output is accomplished in several ways. Suppose `g` is a graph, then `g.set_latex_options()` can be used to efficiently set or modify various options. Setting individual options, or querying options, can be accomplished by first using a command like `opts = g.latex_options()` to obtain a *sage.graphs.graph_latex.GraphLatex* object which has several methods to set and retrieve options.

Here is a minimal session demonstrating how to use these features. The following setup should work in the notebook or at the command-line.:

```
sage: H = graphs.HeawoodGraph()
sage: H.set_latex_options(
....: graphic_size=(5,5),
....: vertex_size=0.2,
....: edge_thickness=0.04,
....: edge_color='green',
....: vertex_color='green',
....: vertex_label_color='red'
....: )
```

At this point, `view(H)` should call `pdflatex` to process the string created by `latex(H)` and then display the resulting graphic.

To use this image in a LaTeX document, you could of course just copy and save the resulting graphic. However, the `latex()` command will produce the underlying LaTeX code, which can be incorporated into a standalone LaTeX document.:

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: latex(H)
\begin{tikzpicture}
\definecolor{cv0}{rgb}{0.0,0.502,0.0}
\definecolor{cfv0}{rgb}{1.0,1.0,1.0}
\definecolor{clv0}{rgb}{1.0,0.0,0.0}
\definecolor{cv1}{rgb}{0.0,0.502,0.0}
\definecolor{cfv1}{rgb}{1.0,1.0,1.0}
\definecolor{clv1}{rgb}{1.0,0.0,0.0}
\definecolor{cv2}{rgb}{0.0,0.502,0.0}
```

(continues on next page)

(continued from previous page)

```
\definecolor{cfv2}{rgb}{1.0,1.0,1.0}
\definecolor{clv2}{rgb}{1.0,0.0,0.0}
\definecolor{cv3}{rgb}{0.0,0.502,0.0}
\definecolor{cfv3}{rgb}{1.0,1.0,1.0}
\definecolor{clv3}{rgb}{1.0,0.0,0.0}
\definecolor{cv4}{rgb}{0.0,0.502,0.0}
\definecolor{cfv4}{rgb}{1.0,1.0,1.0}
\definecolor{clv4}{rgb}{1.0,0.0,0.0}
\definecolor{cv5}{rgb}{0.0,0.502,0.0}
\definecolor{cfv5}{rgb}{1.0,1.0,1.0}
\definecolor{clv5}{rgb}{1.0,0.0,0.0}
\definecolor{cv6}{rgb}{0.0,0.502,0.0}
\definecolor{cfv6}{rgb}{1.0,1.0,1.0}
\definecolor{clv6}{rgb}{1.0,0.0,0.0}
\definecolor{cv7}{rgb}{0.0,0.502,0.0}
\definecolor{cfv7}{rgb}{1.0,1.0,1.0}
\definecolor{clv7}{rgb}{1.0,0.0,0.0}
\definecolor{cv8}{rgb}{0.0,0.502,0.0}
\definecolor{cfv8}{rgb}{1.0,1.0,1.0}
\definecolor{clv8}{rgb}{1.0,0.0,0.0}
\definecolor{cv9}{rgb}{0.0,0.502,0.0}
\definecolor{cfv9}{rgb}{1.0,1.0,1.0}
\definecolor{clv9}{rgb}{1.0,0.0,0.0}
\definecolor{cv10}{rgb}{0.0,0.502,0.0}
\definecolor{cfv10}{rgb}{1.0,1.0,1.0}
\definecolor{clv10}{rgb}{1.0,0.0,0.0}
\definecolor{cv11}{rgb}{0.0,0.502,0.0}
\definecolor{cfv11}{rgb}{1.0,1.0,1.0}
\definecolor{clv11}{rgb}{1.0,0.0,0.0}
\definecolor{cv12}{rgb}{0.0,0.502,0.0}
\definecolor{cfv12}{rgb}{1.0,1.0,1.0}
\definecolor{clv12}{rgb}{1.0,0.0,0.0}
\definecolor{cv13}{rgb}{0.0,0.502,0.0}
\definecolor{cfv13}{rgb}{1.0,1.0,1.0}
\definecolor{clv13}{rgb}{1.0,0.0,0.0}
\definecolor{cv0v1}{rgb}{0.0,0.502,0.0}
\definecolor{cv0v5}{rgb}{0.0,0.502,0.0}
\definecolor{cv0v13}{rgb}{0.0,0.502,0.0}
\definecolor{cv1v2}{rgb}{0.0,0.502,0.0}
\definecolor{cv1v10}{rgb}{0.0,0.502,0.0}
\definecolor{cv2v3}{rgb}{0.0,0.502,0.0}
\definecolor{cv2v7}{rgb}{0.0,0.502,0.0}
\definecolor{cv3v4}{rgb}{0.0,0.502,0.0}
\definecolor{cv3v12}{rgb}{0.0,0.502,0.0}
\definecolor{cv4v5}{rgb}{0.0,0.502,0.0}
\definecolor{cv4v9}{rgb}{0.0,0.502,0.0}
\definecolor{cv5v6}{rgb}{0.0,0.502,0.0}
\definecolor{cv6v7}{rgb}{0.0,0.502,0.0}
\definecolor{cv6v11}{rgb}{0.0,0.502,0.0}
\definecolor{cv7v8}{rgb}{0.0,0.502,0.0}
\definecolor{cv8v9}{rgb}{0.0,0.502,0.0}
\definecolor{cv8v13}{rgb}{0.0,0.502,0.0}
\definecolor{cv9v10}{rgb}{0.0,0.502,0.0}
\definecolor{cv10v11}{rgb}{0.0,0.502,0.0}
\definecolor{cv11v12}{rgb}{0.0,0.502,0.0}
\definecolor{cv12v13}{rgb}{0.0,0.502,0.0}
%
```

(continues on next page)

(continued from previous page)

```
\Vertex[style={minimum size=0.2cm,draw=cv0,fill=cfv0,text=clv0,shape=circle},
↪LabelOut=false,L=\hbox{$0$},x=2.5cm,y=5.0cm]{v0}
\Vertex[style={minimum size=0.2cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},
↪LabelOut=false,L=\hbox{$1$},x=1.3874cm,y=4.7524cm]{v1}
\Vertex[style={minimum size=0.2cm,draw=cv2,fill=cfv2,text=clv2,shape=circle},
↪LabelOut=false,L=\hbox{$2$},x=0.4952cm,y=4.0587cm]{v2}
\Vertex[style={minimum size=0.2cm,draw=cv3,fill=cfv3,text=clv3,shape=circle},
↪LabelOut=false,L=\hbox{$3$},x=0.0cm,y=3.0563cm]{v3}
\Vertex[style={minimum size=0.2cm,draw=cv4,fill=cfv4,text=clv4,shape=circle},
↪LabelOut=false,L=\hbox{$4$},x=0.0cm,y=1.9437cm]{v4}
\Vertex[style={minimum size=0.2cm,draw=cv5,fill=cfv5,text=clv5,shape=circle},
↪LabelOut=false,L=\hbox{$5$},x=0.4952cm,y=0.9413cm]{v5}
\Vertex[style={minimum size=0.2cm,draw=cv6,fill=cfv6,text=clv6,shape=circle},
↪LabelOut=false,L=\hbox{$6$},x=1.3874cm,y=0.2476cm]{v6}
\Vertex[style={minimum size=0.2cm,draw=cv7,fill=cfv7,text=clv7,shape=circle},
↪LabelOut=false,L=\hbox{$7$},x=2.5cm,y=0.0cm]{v7}
\Vertex[style={minimum size=0.2cm,draw=cv8,fill=cfv8,text=clv8,shape=circle},
↪LabelOut=false,L=\hbox{$8$},x=3.6126cm,y=0.2476cm]{v8}
\Vertex[style={minimum size=0.2cm,draw=cv9,fill=cfv9,text=clv9,shape=circle},
↪LabelOut=false,L=\hbox{$9$},x=4.5048cm,y=0.9413cm]{v9}
\Vertex[style={minimum size=0.2cm,draw=cv10,fill=cfv10,text=clv10,shape=circle},
↪LabelOut=false,L=\hbox{$10$},x=5.0cm,y=1.9437cm]{v10}
\Vertex[style={minimum size=0.2cm,draw=cv11,fill=cfv11,text=clv11,shape=circle},
↪LabelOut=false,L=\hbox{$11$},x=5.0cm,y=3.0563cm]{v11}
\Vertex[style={minimum size=0.2cm,draw=cv12,fill=cfv12,text=clv12,shape=circle},
↪LabelOut=false,L=\hbox{$12$},x=4.5048cm,y=4.0587cm]{v12}
\Vertex[style={minimum size=0.2cm,draw=cv13,fill=cfv13,text=clv13,shape=circle},
↪LabelOut=false,L=\hbox{$13$},x=3.6126cm,y=4.7524cm]{v13}
%
\Edge[lw=0.04cm,style={color=cv0v1,},](v0)(v1)
\Edge[lw=0.04cm,style={color=cv0v5,},](v0)(v5)
\Edge[lw=0.04cm,style={color=cv0v13,},](v0)(v13)
\Edge[lw=0.04cm,style={color=cv1v2,},](v1)(v2)
\Edge[lw=0.04cm,style={color=cv1v10,},](v1)(v10)
\Edge[lw=0.04cm,style={color=cv2v3,},](v2)(v3)
\Edge[lw=0.04cm,style={color=cv2v7,},](v2)(v7)
\Edge[lw=0.04cm,style={color=cv3v4,},](v3)(v4)
\Edge[lw=0.04cm,style={color=cv3v12,},](v3)(v12)
\Edge[lw=0.04cm,style={color=cv4v5,},](v4)(v5)
\Edge[lw=0.04cm,style={color=cv4v9,},](v4)(v9)
\Edge[lw=0.04cm,style={color=cv5v6,},](v5)(v6)
\Edge[lw=0.04cm,style={color=cv6v7,},](v6)(v7)
\Edge[lw=0.04cm,style={color=cv6v11,},](v6)(v11)
\Edge[lw=0.04cm,style={color=cv7v8,},](v7)(v8)
\Edge[lw=0.04cm,style={color=cv8v9,},](v8)(v9)
\Edge[lw=0.04cm,style={color=cv8v13,},](v8)(v13)
\Edge[lw=0.04cm,style={color=cv9v10,},](v9)(v10)
\Edge[lw=0.04cm,style={color=cv10v11,},](v10)(v11)
\Edge[lw=0.04cm,style={color=cv11v12,},](v11)(v12)
\Edge[lw=0.04cm,style={color=cv12v13,},](v12)(v13)
%
\end{tikzpicture}
```

EXAMPLES:

This example illustrates switching between the built-in styles when using the tkz_graph format.:

```
sage: g = graphs.PetersenGraph()
sage: g.set_latex_options(tkz_style='Classic')
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: latex(g)
\begin{tikzpicture}
\GraphInit[vstyle=Classic]
...
\end{tikzpicture}
sage: opts = g.latex_options()
sage: opts
LaTeX options for Petersen graph: {'tkz_style': 'Classic'}
sage: g.set_latex_options(tkz_style = 'Art')
sage: opts.get_option('tkz_style')
'Art'
sage: opts
LaTeX options for Petersen graph: {'tkz_style': 'Art'}
sage: latex(g)
\begin{tikzpicture}
\GraphInit[vstyle=Art]
...
\end{tikzpicture}
```

This example illustrates using the optional dot2tex module:

```
sage: g = graphs.PetersenGraph()
sage: g.set_latex_options(format='dot2tex', prog='neato')
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: latex(g)  # optional - dot2tex graphviz
\begin{tikzpicture}[>=latex,line join=bevel,]
...
\end{tikzpicture}
```

Among other things, this supports the flexible `edge_options` option (see *sage.graphs.generic_graph.GenericGraph.graphviz_string()*); here we color in red all edges touching the vertex `0`:

```
sage: g = graphs.PetersenGraph()
sage: g.set_latex_options(format="dot2tex", edge_options=lambda u_v_label: {"color":
↪"red"} if u_v_label[0] == 0 else {})
sage: latex(g)  # optional - dot2tex graphviz
\begin{tikzpicture}[>=latex,line join=bevel,]
...
\end{tikzpicture}
```

5.29.2 GraphLatex class and functions

class sage.graphs.graph_latex.**GraphLatex**(*graph*, ***options*)

Bases: sage.structure.sage_object.SageObject

A class to hold, manipulate and employ options for converting a graph to LaTeX.

This class serves two purposes. First it holds the values of various options designed to work with the `tkz-graph` LaTeX package for rendering graphs. As such, a graph that uses this class will hold a reference to it. Second, this class contains the code to convert a graph into the corresponding LaTeX constructs, returning a string.

EXAMPLES:

```
sage: from sage.graphs.graph_latex import GraphLatex
sage: opts = GraphLatex(graphs.PetersenGraph())
sage: opts
LaTeX options for Petersen graph: {}
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: g == loads(dumps(g))
True
```

dot2tex_picture()

Call `dot2tex` to construct a string of LaTeX commands representing a graph as a `tikzpicture`.

EXAMPLES:

```
sage: g = digraphs.ButterflyGraph(1)
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: print(g.latex_options().dot2tex_picture())  # optional - dot2tex␣
↪graphviz
\begin{tikzpicture}[>=latex,line join=bevel,]
%%
  \node (node_...) at (...bp,...bp) [draw,draw=none] {$\left(...\right)$};
  \node (node_...) at (...bp,...bp) [draw,draw=none] {$\left(...\right)$};
  \node (node_...) at (...bp,...bp) [draw,draw=none] {$\left(...\right)$};
  \node (node_...) at (...bp,...bp) [draw,draw=none] {$\left(...\right)$};
  \draw [black,->] (node_...) ..controls (...bp,...bp) and (...bp,...bp)  ..␣
↪(node_...);
  \draw [black,->] (node_...) ..controls (...bp,...bp) and (...bp,...bp)  ..␣
↪(node_...);
  \draw [black,->] (node_...) ..controls (...bp,...bp) and (...bp,...bp)  ..␣
↪(node_...);
  \draw [black,->] (node_...) ..controls (...bp,...bp) and (...bp,...bp)  ..␣
↪(node_...);
%
\end{tikzpicture}
```

We make sure trac ticket #13624 is fixed:

```
sage: G = DiGraph()
sage: G.add_edge(3333, 88, 'my_label')
sage: G.set_latex_options(edge_labels=True)
sage: print(G.latex_options().dot2tex_picture()) # optional - dot2tex graphviz
\begin{tikzpicture}[>=latex,line join=bevel,]
%%
\node (node_...) at (...bp,...bp) [draw,draw=none] {$...$};
  \node (node_...) at (...bp,...bp) [draw,draw=none] {$...$};
  \draw [black,->] (node_...) ..controls (...bp,...bp) and (...bp,...bp)  ..␣
↪(node_...);
  \definecolor{strokecol}{rgb}{0.0,0.0,0.0};
  \pgfsetstrokecolor{strokecol}
  \draw (...bp,...bp) node {$\text{\texttt{my{\char`\_}label}}$};
%
\end{tikzpicture}
```

Check that trac ticket #25120 is fixed:

```
sage: G = Graph([(0,1)])
sage: G.set_latex_options(edge_colors = {(0,1): 'red'})
sage: print(G.latex_options().dot2tex_picture()) # optional - dot2tex graphviz
\begin{tikzpicture}[>=latex,line join=bevel,]
...
\draw [red,] (node_0) ... (node_1);
...
\end{tikzpicture}
```

Note: There is a lot of overlap between what `tkz_picture` and `dot2tex` do. It would be best to merge them! `dot2tex` probably can work without `graphviz` if layout information is provided.

get_option(*option_name*)

Return the current value of the named option.

INPUT:

- `option_name` – the name of an option

OUTPUT:

If the name is not present in `__graphlatex_options` it is an error to ask for it. If an option has not been set then the default value is returned. Otherwise, the value of the option is returned.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: opts.set_option('tkz_style', 'Art')
sage: opts.get_option('tkz_style')
'Art'
sage: opts.set_option('tkz_style')
sage: opts.get_option('tkz_style') == "Custom"
True
sage: opts.get_option('bad_name')
Traceback (most recent call last):
...
ValueError: bad_name is not a Latex option for a graph.
```

latex()

Return a string in LaTeX representing a graph.

This is the command that is invoked by `sage.graphs.generic_graph.GenericGraph._latex_` for a graph, so it returns a string of LaTeX commands that can be incorporated into a LaTeX document unmodified. The exact contents of this string are influenced by the options set via the methods *sage.graphs.generic_graph.GenericGraph.set_latex_options()*, *set_option()*, and *set_options()*.

By setting the `format` option different packages can be used to create the latex version of a graph. Supported packages are `tkz-graph` and `dot2tex`.

EXAMPLES:

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: g = graphs.CompleteGraph(2)
sage: opts = g.latex_options()
```

(continues on next page)

(continued from previous page)

```
sage: print(opts.latex())
\begin{tikzpicture}
\definecolor{cv0}{rgb}{0.0,0.0,0.0}
\definecolor{cfv0}{rgb}{1.0,1.0,1.0}
\definecolor{clv0}{rgb}{0.0,0.0,0.0}
\definecolor{cv1}{rgb}{0.0,0.0,0.0}
\definecolor{cfv1}{rgb}{1.0,1.0,1.0}
\definecolor{clv1}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v1}{rgb}{0.0,0.0,0.0}
%
\Vertex[style={minimum size=1.0cm,draw=cv0,fill=cfv0,text=clv0,shape=circle},
↪LabelOut=false,L=\hbox{$0$},x=2.5cm,y=5.0cm]{v0}
\Vertex[style={minimum size=1.0cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},
↪LabelOut=false,L=\hbox{$1$},x=2.5cm,y=0.0cm]{v1}
%
\Edge[lw=0.1cm,style={color=cv0v1,},](v0)(v1)
%
\end{tikzpicture}
```

set_option(*option_name*, *option_value=None*)

Set, modify, clear a LaTeX option for controlling the rendering of a graph.

The possible options are documented here, because ultimately it is this routine that sets the values. However, the *sage.graphs.generic_graph.GenericGraph.set_latex_options()* method is the easiest way to set options, and allows several to be set at once.

INPUT:

- `option_name` – a string for a latex option contained in the list `sage.graphs.graph_latex.GraphLatex.__graphlatex_options`. A `ValueError` is raised if the option is not allowed.
- `option_value` – a value for the option. If omitted, or set to `None`, the option will use the default value.

The output can be either handled internally by `Sage`, or delegated to the external software `dot2tex` and `graphviz`. This is controlled by the option `format`:

- `format` – string (default: `'tkz_graph'`); either `'dot2tex'` or `'tkz_graph'`.

If format is `'dot2tex'`, then all the LaTeX generation will be delegated to `dot2tex` (which must be installed).

For `tkz_graph`, the possible option names, and associated values are given below. This first group allows you to set a style for a graph and specify some sizes related to the eventual image. (For more information consult the documentation for the `tkz-graph` package.)

- `tkz_style` – string (default: `'Custom'`); the name of a pre-defined `tkz-graph` style such as `'Shade'`, `'Art'`, `'Normal'`, `'Dijkstra'`, `'Welsh'`, `'Classic'`, and `'Simple'`, or the string `'Custom'`. Using one of these styles alone will often give a reasonably good drawing with minimal effort. For a custom appearance set this to `'Custom'` and use the options described below to override the default values.
- `units` – string (default: `'cm'`) – a natural unit of measurement used for all dimensions. Possible values are: `'in'`, `'mm'`, `'cm'`, `'pt'`, `'em'`, `'ex'`.
- `scale` – float (default: `1.0`); a dimensionless number that multiplies every linear dimension. So you can design at sizes you are accustomed to, then shrink or expand to meet other needs. Though fonts do not scale.

- graphic_size – tuple (default: (5, 5)); overall dimensions (width, length) of the bounding box around the entire graphic image.
- margins – 4-tuple (default: (0, 0, 0, 0)); portion of graphic given over to a plain border as a tuple of four numbers: (left, right, top, bottom). These are subtracted from the graphic_size to create the area left for the vertices of the graph itself. Note that the processing done by Sage will trim the graphic down to the minimum possible size, removing any border. So this is only useful if you use the latex string in a latex document.

If not using a pre-built style the following options are used, so the following defaults will apply. It is not possible to begin with a pre-built style and modify it (other than editing the latex string by hand after the fact).

- vertex_color – (default: 'black'); a single color to use as the default for outline of vertices. For the sphere shape this color is used for the entire vertex, which is drawn with a 3D shading. Colors must be specified as a string recognized by the matplotlib library: a standard color name like 'red', or a hex string like '#2D87A7', or a single character from the choices 'rgbcmykw'. Additionally, a number between 0 and 1 will create a grayscale value. These color specifications are consistent throughout the options for a tikzpicture.
- vertex_colors – a dictionary whose keys are vertices of the graph and whose values are colors. These will be used to color the outline of vertices. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- vertex_fill_color – (default: 'white'); a single color to use as the default for the fill color of vertices. See the explanation above for the vertex_color option to see possible values. This color is ignored for the sphere vertex shape.
- vertex_fill_colors – a dictionary whose keys are vertices of the graph and whose values are colors. These will be used to fill the interior of vertices. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- vertex_shape – string (default: 'circle'); specifies the shape of the vertices. Allowable values are 'circle', 'sphere', 'rectangle', 'diamond'. The sphere shape has a 3D look to its coloring and is uses only one color, that specified by vertex_color and vertex_colors, which are normally used for the outline of the vertex.
- vertex_shapes – a dictionary whose keys are vertices of the graph and whose values are shapes. See vertex_shape for the allowable possibilities.
- vertex_size – float (default: 1.0); the minimum size of a vertex as a number. Vertices will expand to contain their labels if the labels are placed inside the vertices. If you set this value to zero the vertex will be as small as possible (up to tkz-graph's "inner sep" parameter), while still containing labels. However, if labels are not of a uniform size, then the vertices will not be either.
- vertex_sizes – a dictionary of sizes for some of the vertices.
- vertex_labels – boolean (default: True); determine whether or not to display the vertex labels. If False subsequent options about vertex labels are ignored.
- vertex_labels_math – boolean (default: True); when True, if a label is a string that begins and ends with dollar signs, then the string will be rendered as a latex string. Otherwise, the label will be automatically subjected to the latex() method and rendered accordingly. If False the label is rendered as its textual representation according to the _repr method. Support for arbitrarily-complicated mathematics is not especially robust.
- vertex_label_color – (default: 'black'); a single color to use as the default for labels of vertices. See the explanation above for the vertex_color option to see possible values.

- vertex_label_colors – a dictionary whose keys are vertices of the graph and whose values are colors. These will be used for the text of the labels of vertices. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- vertex_label_placement – (default: 'center'); if 'center' the label is centered in the interior of the vertex and the vertex will expand to contain the label. Giving instead a pair of numbers will place the label exterior to the vertex at a certain distance from the edge, and at an angle to the positive x-axis, similar in spirit to polar coordinates.
- vertex_label_placements – a dictionary of placements indexed by the vertices. See the explanation for vertex_label_placement for the possible values.
- edge_color – (default: 'black'); a single color to use as the default for an edge. See the explanation above for the vertex_color option to see possible values.
- edge_colors – a dictionary whose keys are edges of the graph and whose values are colors. These will be used to color the edges. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- edge_fills – boolean (default: False); whether an edge has a second color running down the middle. This can be a useful effect for highlighting edge crossings.
- edge_fill_color – (default: 'black'); a single color to use as the default for the fill color of an edge. The boolean switch edge_fills must be set to True for this to have an effect. See the explanation above for the vertex_color option to see possible values.
- edge_fill_colors – a dictionary whose keys are edges of the graph and whose values are colors. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- edge_thickness – float (default: 0.1); specifies the width of the edges. Note that tkz-graph does not interpret this number for loops.
- edge_thicknesses – a dictionary of thicknesses for some of the edges of a graph. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value.
- edge_labels – boolean (default: False); determine if edge labels are shown. If False subsequent options about edge labels are ignored.
- edge_labels_math – boolean (default: True); control how edge labels are rendered. Read the explanation for the vertex_labels_math option, which behaves identically. Support for arbitrarily-complicated mathematics is not especially robust.
- edge_label_color – (default: 'black'); a single color to use as the default for labels of edges. See the explanation above for the vertex_color option to see possible values.
- edge_label_colors – a dictionary whose keys are edges of the graph and whose values are colors. These will be used for the text of the labels of edges. See the explanation above for the vertex_color option to see possible values. These values need only be specified for a proper subset of the vertices. Specified values will supersede a default value. Note that labels must be used for this to have any effect, and no care is taken to ensure that label and fill colors work well together.
- edge_label_sloped – boolean (default: True); specifies how edge labels are place. False results in a horizontal label, while True means the label is rotated to follow the direction of the edge it labels.
- edge_label_slopes – a dictionary of booleans, indexed by some subset of the edges. See the edge_label_sloped option for a description of sloped edge labels.

- edge_label_placement – (default: 0.50); either a number between 0.0 and 1.0, or one of: 'above', 'below', 'left', 'right'. These adjust the location of an edge label along an edge. A number specifies how far along the edge the label is located. 'left' and 'right' are conveniences. 'above' and 'below' move the label off the edge itself while leaving it near the midpoint of the edge. The default value of 0.50 places the label on the midpoint of the edge.
- edge_label_placements – a dictionary of edge placements, indexed by the edges. See the edge_label_placement option for a description of the allowable values.
- loop_placement – (default: (3.0, 'NO')); determine how loops are rendered. the first element of the pair is a distance, which determines how big the loop is and the second element is a string specifying a compass point (North, South, East, West) as one of 'NO', 'SO', 'EA', 'WE'.
- loop_placements – a dictionary of loop placements. See the loop_placements option for the allowable values. While loops are technically edges, this dictionary is indexed by vertices.

For the 'dot2tex' format, the possible option names and associated values are given below:

- prog – string; the program used for the layout. It must be a string corresponding to one of the software of the graphviz suite: 'dot', 'neato', 'twopi', 'circo' or 'fdp'.
- edge_labels – boolean (default: False); whether to display the labels on edges.
- edge_colors – a color; can be used to set a global color to the edge of the graph.
- color_by_label – boolean (default: False); colors the edges according to their labels
- subgraph_clusters – (default: []) a list of lists of vertices, if supported by the layout engine, nodes belonging to the same cluster subgraph are drawn together, with the entire drawing of the cluster contained within a bounding rectangle.

OUTPUT:

There are none. Success happens silently.

EXAMPLES:

Set, then modify, then clear the tkz_style option, and finally show an error for an unrecognized option name:

```
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: opts
LaTeX options for Petersen graph: {}
sage: opts.set_option('tkz_style', 'Art')
sage: opts
LaTeX options for Petersen graph: {'tkz_style': 'Art'}
sage: opts.set_option('tkz_style', 'Simple')
sage: opts
LaTeX options for Petersen graph: {'tkz_style': 'Simple'}
sage: opts.set_option('tkz_style')
sage: opts
LaTeX options for Petersen graph: {}
sage: opts.set_option('bad_name', 'nonsense')
Traceback (most recent call last):
...
ValueError: bad_name is not a LaTeX option for a graph.
```

See *sage.graphs.generic_graph.GenericGraph.layout_graphviz()* for installation instructions for graphviz and dot2tex. Furthermore, pgf >= 2.00 should be available inside LaTeX's tree for LaTeX compilation (e.g. when using view). In case your LaTeX distribution does not provide it, here are short instructions:

- download pgf from http://sourceforge.net/projects/pgf/
- unpack it in `/usr/share/texmf/tex/generic` (depends on your system)
- clean out remaining pgf files from older version
- run texhash

set_options(***kwds*)

Set several LaTeX options for a graph all at once.

INPUT:

- `kwds` – any number of option/value pairs to set many graph latex options at once (a variable number, in any order). Existing values are overwritten, new values are added. Existing values can be cleared by setting the value to `None`. Errors are raised in the *`set_option()`* method.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: opts = g.latex_options()
sage: opts.set_options(tkz_style='Welsh')
sage: opts.get_option('tkz_style')
'Welsh'
```

tkz_picture()

Return a string of LaTeX commands representing a graph as a `tikzpicture`.

This routine interprets the graph's properties and the options in `_options` to render the graph with commands from the `tkz-graph` LaTeX package.

This requires that the LaTeX optional packages `tkz-graph` and `tkz-berge` be installed. You may also need a current version of the pgf package. If the `tkz-graph` and `tkz-berge` packages are present in the system's TeX installation, the appropriate `\\usepackage{}` commands will be added to the LaTeX preamble as part of the initialization of the graph. If these two packages are not present, then this command will return a warning on its first use, but will return a string that could be used elsewhere, such as a LaTeX document.

For more information about tkz-graph you can visit Altermundus.com

EXAMPLES:

With a pre-built `tkz-graph` style specified, the latex representation will be relatively simple.

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: g = graphs.CompleteGraph(3)
sage: opts = g.latex_options()
sage: g.set_latex_options(tkz_style='Art')
sage: print(opts.tkz_picture())
\begin{tikzpicture}
\GraphInit[vstyle=Art]
%
\Vertex[L=\hbox{$0$},x=2.5cm,y=5.0cm]{v0}
\Vertex[L=\hbox{$1$},x=0.0cm,y=0.0cm]{v1}
\Vertex[L=\hbox{$2$},x=5.0cm,y=0.0cm]{v2}
%
\Edge[](v0)(v1)
\Edge[](v0)(v2)
\Edge[](v1)(v2)
%
\end{tikzpicture}
```

Setting the style to "Custom" results in various configurable aspects set to the defaults, so the string is more involved.

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: g = graphs.CompleteGraph(3)
sage: opts = g.latex_options()
sage: g.set_latex_options(tkz_style='Custom')
sage: print(opts.tkz_picture())
\begin{tikzpicture}
\definecolor{cv0}{rgb}{0.0,0.0,0.0}
\definecolor{cfv0}{rgb}{1.0,1.0,1.0}
\definecolor{clv0}{rgb}{0.0,0.0,0.0}
\definecolor{cv1}{rgb}{0.0,0.0,0.0}
\definecolor{cfv1}{rgb}{1.0,1.0,1.0}
\definecolor{clv1}{rgb}{0.0,0.0,0.0}
\definecolor{cv2}{rgb}{0.0,0.0,0.0}
\definecolor{cfv2}{rgb}{1.0,1.0,1.0}
\definecolor{clv2}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v1}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v2}{rgb}{0.0,0.0,0.0}
\definecolor{cv1v2}{rgb}{0.0,0.0,0.0}
%
\Vertex[style={minimum size=1.0cm,draw=cv0,fill=cfv0,text=clv0,shape=circle},
↪LabelOut=false,L=\hbox{$0$},x=2.5cm,y=5.0cm]{v0}
\Vertex[style={minimum size=1.0cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},
↪LabelOut=false,L=\hbox{$1$},x=0.0cm,y=0.0cm]{v1}
\Vertex[style={minimum size=1.0cm,draw=cv2,fill=cfv2,text=clv2,shape=circle},
↪LabelOut=false,L=\hbox{$2$},x=5.0cm,y=0.0cm]{v2}
%
\Edge[lw=0.1cm,style={color=cv0v1,},](v0)(v1)
\Edge[lw=0.1cm,style={color=cv0v2,},](v0)(v2)
\Edge[lw=0.1cm,style={color=cv1v2,},](v1)(v2)
%
\end{tikzpicture}
```

See the introduction to the *graph_latex* module for more information on the use of this routine.

sage.graphs.graph_latex.**check_tkz_graph**()
Check if the proper LaTeX packages for the `tikzpicture` environment are installed in the user's environment, and issue a warning otherwise.

The warning is only issued on the first call to this function. So any doctest that illustrates the use of the tkz-graph packages should call this once as having random output to exhaust the warnings before testing output.

See also `sage.misc.latex.Latex.check_file()`

sage.graphs.graph_latex.**have_tkz_graph**()
Return `True` if the proper LaTeX packages for the `tikzpicture` environment are installed in the user's environment, namely `tikz`, `tkz-graph` and `tkz-berge`.

The result is cached.

See also `sage.misc.latex.Latex.has_file()`

sage.graphs.graph_latex.**setup_latex_preamble**()
Add appropriate `\usepackage{...}`, and other instructions to the latex preamble for the packages that are needed for processing graphs(`tikz`, `tkz-graph`, `tkz-berge`), if available in the `LaTeX` installation.

See also `sage.misc.latex.Latex.add_package_to_preamble_if_available()`.

EXAMPLES:

```
sage: sage.graphs.graph_latex.setup_latex_preamble()
```

5.30 Graph editor

sage.graphs.graph_editor.**graph_editor**(*graph=None*, *graph_name=None*, *replace_input=True*, ***layout_options*)

Opens a graph editor in the Sage notebook.

INPUT:

- graph - a *Graph* instance (default: graphs.CompleteGraph(2)); the graph to edit
- graph_name - a string (default: None); the variable name to use for the updated instance; by default, this function attempts to determine the name automatically
- replace_input - a boolean (default: True); whether to replace the text in the input cell with the updated graph data when "Save" is clicked; if this is False, the data is **still** evaluated as if it had been entered in the cell

EXAMPLES:

```
sage: g = graphs.CompleteGraph(3)
sage: graph_editor(g)                          # not tested
sage: graph_editor(graphs.HouseGraph())        # not tested
sage: graph_editor(graph_name='my_graph')      # not tested
sage: h = graphs.StarGraph(6)
sage: graph_editor(h, replace_input=False)     # not tested
```

sage.graphs.graph_editor.**graph_to_js**(*g*)

Returns a string representation of a *Graph* instance usable by the *graph_editor()*. The encoded information is the number of vertices, their 2D positions, and a list of edges.

INPUT:

- g - a *Graph* instance

OUTPUT:

- a string

EXAMPLES:

```
sage: from sage.graphs.graph_editor import graph_to_js
sage: G = graphs.CompleteGraph(4)
sage: graph_to_js(G)
'num_vertices=4;edges=[[0,1],[0,2],[0,3],[1,2],[1,3],[2,3]];pos=[[0.5,0.0],[0.0,0.
↪5],[0.5,1.0],[1.0,0.5]];'
sage: graph_to_js(graphs.StarGraph(2))
'num_vertices=3;edges=[[0,1],[0,2]];pos=[[0.0,0.5],[0.0,0.0],[0.0,1.0]];'
```

5.31 Lists of graphs

AUTHORS:

- Robert L. Miller (2007-02-10): initial version

- Emily A. Kirkman (2007-02-13): added show functions (to_graphics_array and show_graphs)

sage.graphs.graph_list.**from_graph6**(*data*)
Return a list of Sage Graphs, given a list of graph6 data.

INPUT:

- `data` – can be a string, a list of strings, or a file stream

EXAMPLES:

```
sage: l = ['N@@?N@UGAGG?gGlKCMO', 'XsGGWOW?CC?C@HQKHqOjYKC_uHWGX?P?~
↪TqIKA`OA@SAOEcEA??']
sage: graphs_list.from_graph6(l)
[Graph on 15 vertices, Graph on 25 vertices]
```

sage.graphs.graph_list.**from_sparse6**(*data*)
Return a list of Sage Graphs, given a list of sparse6 data.

INPUT:

- `data` – can be a string, a list of strings, or a file stream

EXAMPLES:

```
sage: l = [':P_`cBaC_ACd`C_@BC`ABDHaEH_@BF_@CHIK_@BCEHKL_BIKM_BFGHI', ':f`??KO?B_
↪OOSCGE_?OWONDBO?GOJBDO?_SSJdApcOIG`?og_UKEbg?_SKFq@[CCBA`p?
↪oYMFp@gw]Qaa@xEMHDb@hMCBCbQ@ECHEcAKKQKFPOwo[PIDQ{KIHEcQPOkVKEW_
↪WMNKqPWwcRKOOWSKIGCqhWt??___WMJFCahWzEBa`xOu[MpPPKqYNoOOOKHHDBPs|??__
↪gWMKEcAHKgTLErqA?A@a@G{kVLErs?GDBA@XCs\NggWSOJIDbHh@?A@aF']
sage: graphs_list.from_sparse6(l)
[Looped multi-graph on 17 vertices, Looped multi-graph on 39 vertices]
```

sage.graphs.graph_list.**from_whatever**(*data*)
Return a list of Sage Graphs, given a list of whatever kind of data.

INPUT:

- `data` – can be a string, a list/iterable of strings, or a readable file-like object

EXAMPLES:

```
sage: l = ['N@@?N@UGAGG?gGlKCMO', ':P_`cBaC_ACd`C_@BC`ABDHaEH_@BF_@CHIK_@BCEHKL_
↪BIKM_BFGHI']
sage: graphs_list.from_whatever(l)
[Graph on 15 vertices, Looped multi-graph on 17 vertices]
sage: graphs_list.from_whatever('\n'.join(l))
[Graph on 15 vertices, Looped multi-graph on 17 vertices]
```

This example happens to be a mix a sparse and non-sparse graphs, so we don't explicitly put a `.g6` or `.s6` extension, which implies just one or the other:

```
sage: filename = tmp_filename()
sage: with open(filename, 'w') as fobj:
....:     _ = fobj.write('\n'.join(l))
sage: with open(filename) as fobj:
....:     graphs_list.from_whatever(fobj)
[Graph on 15 vertices, Looped multi-graph on 17 vertices]
```

sage.graphs.graph_list.**show_graphs**(*graph_list*, ***kwds*)
Show a maximum of 20 graphs from `graph_list` in a sage graphics array.

If more than 20 graphs are given in the list argument, then it will display one graphics array after another with each containing at most 20 graphs.

Note that to save the image output from the notebook, you must save each graphics array individually. (There will be a small space between graphics arrays).

INPUT:

- `graph_list` – a Python list of Sage Graphs

GRAPH PLOTTING: Defaults to circular layout for graphs. This allows for a nicer display in a small area and takes much less time to compute than the spring-layout algorithm for many graphs.

EXAMPLES: Create a list of graphs:

```
sage: glist = []
sage: glist.append(graphs.CompleteGraph(6))
sage: glist.append(graphs.CompleteBipartiteGraph(4, 5))
sage: glist.append(graphs.BarbellGraph(7, 4))
sage: glist.append(graphs.CycleGraph(15))
sage: glist.append(graphs.DiamondGraph())
sage: glist.append(graphs.HouseGraph())
sage: glist.append(graphs.HouseXGraph())
sage: glist.append(graphs.KrackhardtKiteGraph())
sage: glist.append(graphs.LadderGraph(5))
sage: glist.append(graphs.LollipopGraph(5, 6))
sage: glist.append(graphs.PathGraph(15))
sage: glist.append(graphs.PetersenGraph())
sage: glist.append(graphs.StarGraph(17))
sage: glist.append(graphs.WheelGraph(9))
```

Check that length is <= 20:

```
sage: len(glist)
14
```

Show the graphs in a graphics array:

```
sage: graphs_list.show_graphs(glist)
```

Example where more than one graphics array is used:

```
sage: gq = GraphQuery(display_cols=['graph6'], num_vertices=5)
sage: g = gq.get_graphs_list()
sage: len(g)
34
sage: graphs_list.show_graphs(g)
```

See the .plot() or .show() documentation for an individual graph for options, all of which are available from *to_graphics_array()*:

```
sage: glist = []
sage: for _ in range(10):
....:     glist.append(graphs.RandomLobster(41, .3, .4))
sage: graphs_list.show_graphs(glist, layout='spring', vertex_size=20)
```

`sage.graphs.graph_list.`**`to_graph6`**(*graphs*, *file=None*, *output_list=False*)

Convert a list of Sage graphs to a single string of graph6 graphs.

If `file` is specified, then the string will be written quietly to the file. If `output_list` is `True`, then a list of strings will be returned, one string per graph.

INPUT:

- `graphs` – a Python list of Sage Graphs
- `file` – (optional) a file stream to write to (must be in 'w' mode)
- `output_list` – boolean (default: `False`); whether to return a string (when set to `True`) or a list of strings. This parameter is ignored if file gets specified

EXAMPLES:

```
sage: l = [graphs.DodecahedralGraph(), graphs.PetersenGraph()]
sage: graphs_list.to_graph6(l)
'ShCHGD@?K?_@?@?C_GGG@??cG?G?GK_?C\nIheA@GUAo\n'
```

sage.graphs.graph_list.**to_graphics_array**(*graph_list*, ***kwds*)
Draw all graphs in a graphics array

INPUT:

- `graph_list` – a Python list of Sage Graphs

GRAPH PLOTTING:

Defaults to circular layout for graphs. This allows for a nicer display in a small area and takes much less time to compute than the spring- layout algorithm for many graphs.

EXAMPLES:

```
sage: glist = []
sage: for i in range(999):
....:     glist.append(graphs.RandomGNP(6, .45))
sage: garray = graphs_list.to_graphics_array(glist)
sage: garray.nrows(), garray.ncols()
(250, 4)
```

See the .plot() or .show() documentation for an individual graph for options, all of which are available from *to_graphics_array()*:

```
sage: glist = []
sage: for _ in range(10):
....:     glist.append(graphs.RandomLobster(41, .3, .4))
sage: graphs_list.to_graphics_array(glist, layout='spring', vertex_size=20)
Graphics Array of size 3 x 4
```

sage.graphs.graph_list.**to_sparse6**(*graphs*, *file=None*, *output_list=False*)
Convert a list of Sage graphs to a single string of sparse6 graphs.

If `file` is specified, then the string will be written quietly to the file. If `output_list` is `True`, then a list of strings will be returned, one string per graph.

INPUT:

- `graphs` – a Python list of Sage Graphs
- `file` – (optional) a file stream to write to (must be in 'w' mode)
- `output_list` – boolean (default: `False`); whether to return a string (when set to `True`) or a list of strings. This parameter is ignored if file gets specified

EXAMPLES:

```
sage: l = [graphs.DodecahedralGraph(), graphs.PetersenGraph()]
sage: graphs_list.to_sparse6(l)
':S_`abcaDe`Fg_HijhKfLdMkNcOjP_BQ\n:I`ES@obGkqegW~\n'
```

5.32 Functions for reading/building graphs/digraphs.

This module gathers functions needed to build a graph from any other data.

Note: This is an **internal** module of Sage. All features implemented here are made available to end-users through the constructors of *Graph* and *DiGraph*.

Note that because they are called by the constructors of *Graph* and *DiGraph*, most of these functions modify a graph inplace.

from_adjacency_matrix()	Fill G with the data of an adjacency matrix.
from_dict_of_dicts()	Fill G with the data of a dictionary of dictionaries.
from_dict_of_lists()	Fill G with the data of a dictionary of lists.
from_dig6()	Fill G with the data of a dig6 string.
from_graph6()	Fill G with the data of a graph6 string.
from_incidence_matrix()	Fill G with the data of an incidence matrix.
from_oriented_incidence_matrix()	Fill G with the data of an *oriented* incidence matrix.
from_seidel_adjacency_matrix()	Fill G with the data of a Seidel adjacency matrix.
from_sparse6()	Fill G with the data of a sparse6 string.

5.32.1 Functions

sage.graphs.graph_input.**from_adjacency_matrix**(*G*, *M*, *loops=False*, *multiedges=False*, *weighted=False*)

Fill G with the data of an adjacency matrix.

INPUT:

- G – a *Graph* or *DiGraph*
- M – an adjacency matrix
- loops, multiedges, weighted – booleans (default: False); whether to consider the graph as having loops, multiple edges, or weights

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_adjacency_matrix
sage: g = Graph()
sage: from_adjacency_matrix(g, graphs.PetersenGraph().adjacency_matrix())
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_dict_of_dicts**(*G*, *M*, *loops=False*, *multiedges=False*, *weighted=False*, *convert_empty_dict_labels_to_None=False*)

Fill G with the data of a dictionary of dictionaries.

INPUT:

- G – a graph
- M – a dictionary of dictionaries
- loops, multiedges, weighted – booleans (default: False); whether to consider the graph as having loops, multiple edges, or weights
- convert_empty_dict_labels_to_None – booleans (default: False); whether to adjust for empty dicts instead of None in NetworkX default edge labels

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_dict_of_dicts
sage: g = Graph()
sage: from_dict_of_dicts(g, graphs.PetersenGraph().to_dictionary(edge_
↪labels=True))
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_dict_of_lists**(*G*, *D*, *loops=False*, *multiedges=False*, *weighted=False*)

Fill G with the data of a dictionary of lists.

INPUT:

- G – a *Graph* or *DiGraph*
- D – a dictionary of lists
- loops, multiedges, weighted – booleans (default: False); whether to consider the graph as having loops, multiple edges, or weights

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_dict_of_lists
sage: g = Graph()
sage: from_dict_of_lists(g, graphs.PetersenGraph().to_dictionary())
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_dig6**(*G*, *dig6_string*)

Fill G with the data of a dig6 string.

INPUT:

- G – a graph
- dig6_string – a dig6 string

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_dig6
sage: g = DiGraph()
sage: from_dig6(g, digraphs.Circuit(10).dig6_string())
sage: g.is_isomorphic(digraphs.Circuit(10))
True
```

sage.graphs.graph_input.**from_graph6**(*G*, *g6_string*)

Fill G with the data of a graph6 string.

INPUT:

- G – a graph

- g6_string – a graph6 string

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_graph6
sage: g = Graph()
sage: from_graph6(g, 'IheA@GUAo')
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_incidence_matrix**(*G*, *M*, *loops=False*, *multiedges=False*, *weighted=False*)

Fill G with the data of an incidence matrix.

INPUT:

- G – a graph
- M – an incidence matrix
- loops, multiedges, weighted – booleans (default: False); whether to consider the graph as having loops, multiple edges, or weights

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_incidence_matrix
sage: g = Graph()
sage: from_incidence_matrix(g, graphs.PetersenGraph().incidence_matrix())
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_oriented_incidence_matrix**(*G*, *M*, *loops=False*, *multiedges=False*, *weighted=False*)

Fill G with the data of an *oriented* incidence matrix.

An oriented incidence matrix is the incidence matrix of a directed graph, in which each non-loop edge corresponds to a $+1$ and a -1, indicating its source and destination.

INPUT:

- G – a *DiGraph*
- M – an incidence matrix
- loops, multiedges, weighted – booleans (default: False); whether to consider the graph as having loops, multiple edges, or weights

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_oriented_incidence_matrix
sage: g = DiGraph()
sage: from_oriented_incidence_matrix(g, digraphs.Circuit(10).incidence_matrix())
sage: g.is_isomorphic(digraphs.Circuit(10))
True
```

sage.graphs.graph_input.**from_seidel_adjacency_matrix**(*G*, *M*)

Fill G with the data of a Seidel adjacency matrix.

INPUT:

- G – a graph
- M – a Seidel adjacency matrix

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_seidel_adjacency_matrix
sage: g = Graph()
sage: from_seidel_adjacency_matrix(g, graphs.PetersenGraph().seidel_adjacency_
↪matrix())
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

sage.graphs.graph_input.**from_sparse6**(*G*, *g6_string*)

Fill G with the data of a sparse6 string.

INPUT:

- G – a graph
- g6_string – a sparse6 string

EXAMPLES:

```
sage: from sage.graphs.graph_input import from_sparse6
sage: g = Graph()
sage: from_sparse6(g, ':I`ES@obGkqegW~')
sage: g.is_isomorphic(graphs.PetersenGraph())
True
```

5.33 Hyperbolicity

Definition :

The hyperbolicity δ of a graph G has been defined by Gromov [Gro1987] as follows (we give here the so-called 4-points condition):

Let a, b, c, d be vertices of the graph, let S_1, S_2 and S_3 be defined by

$$S_1 = dist(a, b) + dist(d, c)$$
$$S_2 = dist(a, c) + dist(b, d)$$
$$S_3 = dist(a, d) + dist(b, c)$$

and let M_1 and M_2 be the two largest values among S_1, S_2, and S_3. We define $hyp(a, b, c, d) = M_1 - M_2$, and the hyperbolicity $\delta(G)$ of the graph is the maximum of hyp over all possible 4-tuples (a, b, c, d) divided by 2. That is, the graph is said δ-hyperbolic when

$$\delta(G) = \frac{1}{2} \max_{a,b,c,d \in V(G)} hyp(a, b, c, d)$$

(note that $hyp(a, b, c, d) = 0$ whenever two elements among a, b, c, d are equal)

Some known results :

- Trees and cliques are 0-hyperbolic
- $n \times n$ grids are $n - 1$-hyperbolic
- Cycles are approximately $n/4$-hyperbolic
- Chordal graphs are ≤ 1-hyperbolic

Besides, the hyperbolicity of a graph is the maximum over all its biconnected components.

Algorithms and complexity :

The time complexity of the naive implementation (i.e. testing all 4-tuples) is $O(n^4)$, and an algorithm with time complexity $O(n^{3.69})$ has been proposed in [FIV2012]. This remains very long for large-scale graphs, and much harder to implement.

Several improvements over the naive algorithm have been proposed and are implemented in the current module.

- Another upper bound on $hyp(a,b,c,d)$ has been proved in [CCL2015]. It is used to design an algorithm with worse case time complexity in $O(n^4)$ but that behaves much better in practice.

  Assume that $S_1 = dist(a,b) + dist(c,d)$ is the largest sum among S_1, S_2, S_3. We have

  $$\begin{aligned} S_2 + S_3 =& dist(a,c) + dist(b,d) + dist(a,d) + dist(b,c) \\ =& [dist(a,c) + dist(b,c)] + [dist(a,d) + dist(b,d)] \\ \geq& dist(a,b) + dist(a,b) \\ \geq& 2dist(a,b) \end{aligned}$$

  Now, since S_1 is the largest sum, we have

  $$\begin{aligned} hyp(a,b,c,d) =& S_1 - \max\{S_2, S_3\} \\ \leq& S_1 - \frac{S_2 + S_3}{2} \\ \leq& S_1 - dist(a,b) \\ =& dist(c,d) \end{aligned}$$

  We obtain similarly that $hyp(a,b,c,d) \leq dist(a,b)$. Consequently, in the implementation of the 'CCL' algorithm, we ensure that S_1 is larger than S_2 and S_3 using an ordering of the pairs by decreasing lengths. Then, we use the best value h found so far to stop exploration as soon as $dist(a,b) \leq h$.

  The worst case time complexity of this algorithm is $O(n^4)$, but it performs very well in practice since it cuts the search space. This algorithm can be turned into an approximation algorithm since at any step of its execution we maintain an upper and a lower bound. We can thus stop execution as soon as a multiplicative approximation factor or an additive one is proven.

- The notion of ''far-apart pairs" has been introduced in [Sot2011] to further reduce the number of 4-tuples to consider. We say that the pair (a,b) is far-apart if for every w in $V \setminus \{a,b\}$ we have

  $$dist(w,a) + dist(a,b) > dist(w,b) \text{ and } dist(w,b) + dist(a,b) > dist(w,a)$$

  Determining the set of far-apart pairs can be done in time $O(nm)$ using BFS. Now, it is proved in [Sot2011] that there exists two far-apart pairs (a,b) and (c,d) satisfying $\delta(G) = hyp(a,b,c,d)/2$. For instance, the $n \times m$-grid has only two far-apart pairs, and so computing its hyperbolicity is immediate once the far-apart pairs are found. The 'CCL+FA' or 'CCL+' algorithm improves the 'CCL' algorithm since it uses far-apart pairs.

- This algorithm was further improved in [BCCM2015]: instead of iterating twice over all pairs of vertices, in the "inner" loop, we cut several pairs by exploiting properties of the underlying graph.

Todo:

- Add exact methods for the hyperbolicity of chordal graphs
- Add method for partitioning the graph with clique separators

This module contains the following functions

At Python level :

hyperbolicity()	Return the hyperbolicity of the graph or an approximation of this value.
hyperbolicity_distribution()	Return the hyperbolicity distribution of the graph or a sampling of it.

AUTHORS:

- David Coudert (2012): initial version, exact and approximate algorithm, distribution, sampling
- David Coudert (2014): improved exact algorithm using far-apart pairs
- Michele Borassi (2015): cleaned the code and implemented the new algorithm
- Karan Desai (2016): fixed minor typo in documentation

5.33.1 Methods

sage.graphs.hyperbolicity.**hyperbolicity**(*G*, *algorithm='BCCM'*, *approximation_factor=None*, *additive_gap=None*, *verbose=False*)

Returns the hyperbolicity of the graph or an approximation of this value.

The hyperbolicity of a graph has been defined by Gromov [Gro1987] as follows: Let a, b, c, d be vertices of the graph, let $S_1 = dist(a, b) + dist(b, c)$, $S_2 = dist(a, c) + dist(b, d)$, and $S_3 = dist(a, d) + dist(b, c)$, and let M_1 and M_2 be the two largest values among S_1, S_2, and S_3. We have $hyp(a, b, c, d) = |M_1 - M_2|$, and the hyperbolicity of the graph is the maximum over all possible 4-tuples (a, b, c, d) divided by 2. The worst case time complexity is in $O(n^4)$.

See the documentation of *sage.graphs.hyperbolicity* for more information.

INPUT:

- `G` – a connected Graph
- `algorithm` – (default: `'BCCM'`); specifies the algorithm to use among:
  - `'basic'` is an exhaustive algorithm considering all possible 4-tuples and so have time complexity in $O(n^4)$.
  - `'CCL'` is an exact algorithm proposed in [CCL2015]. It considers the 4-tuples in an ordering allowing to cut the search space as soon as a new lower bound is found (see the module's documentation). This algorithm can be turned into a approximation algorithm.
  - `'CCL+FA'` or `'CCL+'` uses the notion of far-apart pairs as proposed in [Sot2011] to significantly reduce the overall computation time of the `'CCL'` algorithm.
  - `'BCCM'` is an exact algorithm proposed in [BCCM2015]. It improves `'CCL+FA'` by cutting several 4-tuples (for more information, see the module's documentation).
  - `'dom'` is an approximation with additive constant four. It computes the hyperbolicity of the vertices of a dominating set of the graph. This is sometimes slower than `'CCL'` and sometimes faster. Try it to know if it is interesting for you. The `additive_gap` and `approximation_factor` parameters cannot be used in combination with this method and so are ignored.
- `approximation_factor` – (default: None) When the approximation factor is set to some value (larger than 1.0), the function stop computations as soon as the ratio between the upper bound and the best found

solution is less than the approximation factor. When the approximation factor is 1.0, the problem is solved optimally. This parameter is used only when the chosen algorithm is `'CCL'`, `'CCL+FA'`, or `'BCCM'`.

- `additive_gap` – (default: None) When sets to a positive number, the function stop computations as soon as the difference between the upper bound and the best found solution is less than additive gap. When the gap is 0.0, the problem is solved optimally. This parameter is used only when the chosen algorithm is `'CCL'` or `'CCL+FA'`, or `'BCCM'`.
- `verbose` – (default: `False`) is a boolean set to True to display some information during execution: new upper and lower bounds, etc.

OUTPUT:

This function returns the tuple (delta, certificate, delta_UB), where:

- `delta` – the hyperbolicity of the graph (half-integer value).
- `certificate` – is the list of the 4 vertices for which the maximum value has been computed, and so the hyperbolicity of the graph.
- `delta_UB` – is an upper bound for `delta`. When `delta == delta_UB`, the returned solution is optimal. Otherwise, the approximation factor if `delta_UB/delta`.

EXAMPLES:

Hyperbolicity of a 3×3 grid:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.Grid2dGraph(3, 3)
sage: L,C,U = hyperbolicity(G, algorithm='BCCM'); L,sorted(C),U
(2, [(0, 0), (0, 2), (2, 0), (2, 2)], 2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL'); L,sorted(C),U
(2, [(0, 0), (0, 2), (2, 0), (2, 2)], 2)
sage: L,C,U = hyperbolicity(G, algorithm='basic'); L,sorted(C),U
(2, [(0, 0), (0, 2), (2, 0), (2, 2)], 2)
```

Hyperbolicity of a PetersenGraph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.PetersenGraph()
sage: L,C,U = hyperbolicity(G, algorithm='BCCM'); L,sorted(C),U
(1/2, [6, 7, 8, 9], 1/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL'); L,sorted(C),U
(1/2, [0, 1, 2, 3], 1/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL+'); L,sorted(C),U
(1/2, [0, 1, 2, 3], 1/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL+FA'); L,sorted(C),U
(1/2, [0, 1, 2, 3], 1/2)
sage: L,C,U = hyperbolicity(G, algorithm='basic'); L,sorted(C),U
(1/2, [0, 1, 2, 3], 1/2)
sage: L,C,U = hyperbolicity(G, algorithm='dom'); L,sorted(C),U
(0, [0, 1, 2, 6], 1)
```

Asking for an approximation in a grid graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.Grid2dGraph(2, 10)
sage: L,C,U = hyperbolicity(G, algorithm='CCL', approximation_factor=1.5); L,U
(1, 3/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL+', approximation_factor=1.5); L,U
(1, 1)
```

(continues on next page)

(continued from previous page)

```
sage: L,C,U = hyperbolicity(G, algorithm='CCL', approximation_factor=4); L,U
(1, 4)
sage: L,C,U = hyperbolicity(G, algorithm='CCL', additive_gap=2); L,U
(1, 3)
sage: L,C,U = hyperbolicity(G, algorithm='dom'); L,U
(1, 5)
```

Asking for an approximation in a cycle graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.CycleGraph(10)
sage: L,C,U = hyperbolicity(G, algorithm='CCL', approximation_factor=1.5); L,U
(2, 5/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL+FA', approximation_factor=1.5); L,U
(2, 5/2)
sage: L,C,U = hyperbolicity(G, algorithm='CCL+FA', additive_gap=1); L,U
(2, 5/2)
```

Comparison of results:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: for i in range(10): # long time
....:     G = graphs.RandomBarabasiAlbert(100,2)
....:     d1,_,_ = hyperbolicity(G, algorithm='basic')
....:     d2,_,_ = hyperbolicity(G, algorithm='CCL')
....:     d3,_,_ = hyperbolicity(G, algorithm='CCL+')
....:     d4,_,_ = hyperbolicity(G, algorithm='CCL+FA')
....:     d5,_,_ = hyperbolicity(G, algorithm='BCCM')
....:     l3,_,u3 = hyperbolicity(G, approximation_factor=2)
....:     if (not d1==d2==d3==d4==d5) or l3>d1 or u3<d1:
....:        print("That's not good!")

sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: import random
sage: random.seed()
sage: for i in range(10): # long time
....:     n = random.randint(2, 20)
....:     m = random.randint(0, n*(n-1) / 2)
....:     G = graphs.RandomGNM(n, m)
....:     for cc in G.connected_components_subgraphs():
....:         d1,_,_ = hyperbolicity(cc, algorithm='basic')
....:         d2,_,_ = hyperbolicity(cc, algorithm='CCL')
....:         d3,_,_ = hyperbolicity(cc, algorithm='CCL+')
....:         d4,_,_ = hyperbolicity(cc, algorithm='CCL+FA')
....:         d5,_,_ = hyperbolicity(cc, algorithm='BCCM')
....:         l3,_,u3 = hyperbolicity(cc, approximation_factor=2)
....:         if (not d1==d2==d3==d4==d5) or l3>d1 or u3<d1:
....:             print("Error in graph ", cc.edges())
```

The hyperbolicity of a graph is the maximum value over all its biconnected components:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.PetersenGraph() * 2
sage: G.add_edge(0, 11)
sage: L,C,U = hyperbolicity(G); L,sorted(C),U
(1/2, [6, 7, 8, 9], 1/2)
```

sage.graphs.hyperbolicity.**hyperbolicity_distribution**(*G*, *algorithm='sampling'*, *sampling_size=1000000*)

Return the hyperbolicity distribution of the graph or a sampling of it.

The hyperbolicity of a graph has been defined by Gromov [Gro1987] as follows: Let a, b, c, d be vertices of the graph, let $S_1 = dist(a, b) + dist(b, c)$, $S_2 = dist(a, c) + dist(b, d)$, and $S_3 = dist(a, d) + dist(b, c)$, and let M_1 and M_2 be the two largest values among S_1, S_2, and S_3. We have $hyp(a, b, c, d) = |M_1 - M_2|$, and the hyperbolicity of the graph is the maximum over all possible 4-tuples (a, b, c, d) divided by 2.

The computation of the hyperbolicity of each 4-tuple, and so the hyperbolicity distribution, takes time in $O(n^4)$.

INPUT:

- `G` – a Graph.
- `algorithm` – (default: 'sampling') When algorithm is 'sampling', it returns the distribution of the hyperbolicity over a sample of `sampling_size` 4-tuples. When algorithm is 'exact', it computes the distribution of the hyperbolicity over all 4-tuples. Be aware that the computation time can be HUGE.
- `sampling_size` – (default: 10^6) number of 4-tuples considered in the sampling. Used only when `algorithm == 'sampling'`.

OUTPUT:

- `hdict` – A dictionary such that hdict[i] is the number of 4-tuples of hyperbolicity i.

EXAMPLES:

Exact hyperbolicity distribution of the Petersen Graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity_distribution
sage: G = graphs.PetersenGraph()
sage: hyperbolicity_distribution(G,algorithm='exact')
{0: 3/7, 1/2: 4/7}
```

Exact hyperbolicity distribution of a 3×3 grid:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity_distribution
sage: G = graphs.GridGraph([3,3])
sage: hyperbolicity_distribution(G,algorithm='exact')
{0: 11/18, 1: 8/21, 2: 1/126}
```

5.34 Tutte polynomial

This module implements a deletion-contraction algorithm for computing the Tutte polynomial as described in the paper [HPR2010].

tutte_polynomial()	Computes the Tutte polynomial of the input graph

Authors:

- Mike Hansen (06-2013), Implemented the algorithm.
- Jernej Azarija (06-2013), Tweaked the code, added documentation

5.34.1 Definition

Given a graph G, with n vertices and m edges and $k(G)$ connected components we define the Tutte polynomial of G as

$$\sum_H (x-1)^{k(H)-c}(y-1)^{k(H)-|E(H)|-n}$$

where the sum ranges over all induced subgraphs H of G.

5.34.2 Functions

class sage.graphs.tutte_polynomial.**Ear**(*graph*, *end_points*, *interior*, *is_cycle*)

Bases: object

An ear is a sequence of vertices

Here is the definition from [HPR2010]:

An ear in a graph is a path $v_1 - v_2 - \cdots - v_n - v_{n+1}$ where $d(v_1) > 2$, $d(v_{n+1}) > 2$ and $d(v_2) = d(v_3) = \cdots = d(v_n) = 2$.

A cycle is viewed as a special ear where $v_1 = v_{n+1}$ and the restriction on the degree of this vertex is lifted.

INPUT:

static find_ear(*g*)

Finds the first ear in a graph.

EXAMPLES:

```
sage: G = graphs.PathGraph(4)
sage: G.add_edges([(0,4),(0,5),(3,6),(3,7)])
sage: from sage.graphs.tutte_polynomial import Ear
sage: E = Ear.find_ear(G)
sage: E.s
3
sage: E.unlabeled_edges
[(0, 1), (1, 2), (2, 3)]
sage: E.vertices
[0, 1, 2, 3]
```

removed_from(**args*, ***kwds*)

A context manager which removes the ear from the graph G.

EXAMPLES:

```
sage: G = graphs.PathGraph(4)
sage: G.add_edges([(0,4),(0,5),(3,6),(3,7)])
sage: len(G.edges())
7
sage: from sage.graphs.tutte_polynomial import Ear
sage: E = Ear.find_ear(G)
sage: with E.removed_from(G) as Y:
....:     G.edges()
[(0, 4, None), (0, 5, None), (3, 6, None), (3, 7, None)]
sage: len(G.edges())
7
```

s

Returns the number of distinct edges in this ear.

EXAMPLES:

```
sage: G = graphs.PathGraph(4)
sage: G.add_edges([(0,4),(0,5),(3,6),(3,7)])
sage: from sage.graphs.tutte_polynomial import Ear
sage: E = Ear(G,[0,3],[1,2],False)
sage: E.s
3
```

unlabeled_edges()

Returns the edges in this ear.

EXAMPLES:

```
sage: G = graphs.PathGraph(4)
sage: G.add_edges([(0,4),(0,5),(3,6),(3,7)])
sage: from sage.graphs.tutte_polynomial import Ear
sage: E = Ear(G,[0,3],[1,2],False)
sage: E.unlabeled_edges
[(0, 1), (1, 2), (2, 3)]
```

vertices

Returns the vertices of this ear.

EXAMPLES:

```
sage: G = graphs.PathGraph(4)
sage: G.add_edges([(0,4),(0,5),(3,6),(3,7)])
sage: from sage.graphs.tutte_polynomial import Ear
sage: E = Ear(G,[0,3],[1,2],False)
sage: E.vertices
[0, 1, 2, 3]
```

class sage.graphs.tutte_polynomial.**EdgeSelection**

Bases: object

class sage.graphs.tutte_polynomial.**MaximizeDegree**

Bases: *sage.graphs.tutte_polynomial.EdgeSelection*

class sage.graphs.tutte_polynomial.**MinimizeDegree**

Bases: *sage.graphs.tutte_polynomial.EdgeSelection*

class sage.graphs.tutte_polynomial.**MinimizeSingleDegree**

Bases: *sage.graphs.tutte_polynomial.EdgeSelection*

class sage.graphs.tutte_polynomial.**VertexOrder**(*order*)

Bases: *sage.graphs.tutte_polynomial.EdgeSelection*

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import VertexOrder
sage: A = VertexOrder([4,6,3,2,1,7])
sage: A.order
[4, 6, 3, 2, 1, 7]
sage: A.inverse_order
{1: 4, 2: 3, 3: 2, 4: 0, 6: 1, 7: 5}
```

sage.graphs.tutte_polynomial.**contracted_edge**(**args*, ***kwds*)

Delete the first vertex in the edge, and make all the edges that went from it go to the second vertex.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import contracted_edge
sage: G = Graph(multiedges=True)
sage: G.add_edges([(0,1,'a'),(1,2,'b'),(0,3,'c')])
sage: G.edges()
[(0, 1, 'a'), (0, 3, 'c'), (1, 2, 'b')]
sage: with contracted_edge(G,(0,1)) as Y:
....:     G.edges(); G.vertices()
[(1, 2, 'b'), (1, 3, 'c')]
[1, 2, 3]
sage: G.edges()
[(0, 1, 'a'), (0, 3, 'c'), (1, 2, 'b')]
```

sage.graphs.tutte_polynomial.**edge_multiplicities**(*G*)

Return the dictionary of multiplicities of the edges in the graph G.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import edge_multiplicities
sage: G = Graph({1: [2,2,3], 2: [2], 3: [4,4], 4: [2,2,2]})
sage: sorted(edge_multiplicities(G).items())
[((1, 2), 2), ((1, 3), 1), ((2, 2), 1), ((2, 4), 3), ((3, 4), 2)]
```

sage.graphs.tutte_polynomial.**removed_edge**(**args*, ***kwds*)

A context manager which removes an edge from the graph G and restores it upon exiting.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import removed_edge
sage: G = Graph()
sage: G.add_edge(0,1)
sage: G.edges()
[(0, 1, None)]
sage: with removed_edge(G,(0,1)) as Y:
....:     G.edges(); G.vertices()
[]
[0, 1]
sage: G.edges()
[(0, 1, None)]
```

sage.graphs.tutte_polynomial.**removed_loops**(**args*, ***kwds*)

A context manager which removes all the loops in the graph G. It yields a list of the loops, and restores the loops upon exiting.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import removed_loops
sage: G = Graph(multiedges=True, loops=True)
sage: G.add_edges([(0,1,'a'),(1,2,'b'),(0,0,'c')])
sage: G.edges()
[(0, 0, 'c'), (0, 1, 'a'), (1, 2, 'b')]
sage: with removed_loops(G) as Y:
....:     G.edges(); G.vertices(); Y
[(0, 1, 'a'), (1, 2, 'b')]
[0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
[(0, 0, 'c')]
sage: G.edges()
[(0, 0, 'c'), (0, 1, 'a'), (1, 2, 'b')]
```

sage.graphs.tutte_polynomial.**removed_multiedge**(**args*, ***kwds*)

A context manager which removes an edge with multiplicity from the graph G and restores it upon exiting.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import removed_multiedge
sage: G = Graph(multiedges=True)
sage: G.add_edges([(0,1,'a'),(0,1,'b')])
sage: G.edges()
[(0, 1, 'a'), (0, 1, 'b')]
sage: with removed_multiedge(G,(0,1)) as Y:
....:     G.edges()
[]
sage: G.edges()
[(0, 1, 'a'), (0, 1, 'b')]
```

sage.graphs.tutte_polynomial.**tutte_polynomial**(*G*, *edge_selector=None*, *cache=None*)

Return the Tutte polynomial of the graph G.

INPUT:

- `edge_selector` (optional; method) this argument allows the user to specify his own heuristic for selecting edges used in the deletion contraction recurrence
- `cache` – (optional; dict) a dictionary to cache the Tutte polynomials generated in the recursive process. One will be created automatically if not provided.

EXAMPLES:

The Tutte polynomial of any tree of order n is x^{n-1}:

```
sage: all(T.tutte_polynomial() == x**9 for T in graphs.trees(10))
True
```

The Tutte polynomial of the Petersen graph is:

```
sage: P = graphs.PetersenGraph()
sage: P.tutte_polynomial()
x^9 + 6*x^8 + 21*x^7 + 56*x^6 + 12*x^5*y + y^6 + 114*x^5 + 70*x^4*y
+ 30*x^3*y^2 + 15*x^2*y^3 + 10*x*y^4 + 9*y^5 + 170*x^4 + 170*x^3*y
+ 105*x^2*y^2 + 65*x*y^3 + 35*y^4 + 180*x^3 + 240*x^2*y + 171*x*y^2
+ 75*y^3 + 120*x^2 + 168*x*y + 84*y^2 + 36*x + 36*y
```

The Tutte polynomial of G evaluated at (1,1) is the number of spanning trees of G:

```
sage: G = graphs.RandomGNP(10,0.6)
sage: G.tutte_polynomial()(1,1) == G.spanning_trees_count()
True
```

Given that $T(x, y)$ is the Tutte polynomial of a graph G with n vertices and c connected components, then $(-1)^{n-c}x^kT(1-x, 0)$ is the chromatic polynomial of G.

```
sage: G = graphs.OctahedralGraph()
sage: T = G.tutte_polynomial()
```

(continues on next page)

(continued from previous page)

```
sage: R = PolynomialRing(ZZ, 'x')
sage: R((-1)^5*x*T(1-x,0)).factor()
(x - 2) * (x - 1) * x * (x^3 - 9*x^2 + 29*x - 32)
sage: G.chromatic_polynomial().factor()
(x - 2) * (x - 1) * x * (x^3 - 9*x^2 + 29*x - 32)
```

sage.graphs.tutte_polynomial.**underlying_graph**(G)

Given a graph G with multi-edges, returns a graph where all the multi-edges are replaced with a single edge.

EXAMPLES:

```
sage: from sage.graphs.tutte_polynomial import underlying_graph
sage: G = Graph(multiedges=True)
sage: G.add_edges([(0,1,'a'),(0,1,'b')])
sage: G.edges()
[(0, 1, 'a'), (0, 1, 'b')]
sage: underlying_graph(G).edges()
[(0, 1, None)]
```

5.35 Partial cubes

The code in this module that recognizes partial cubes is originally from the PADS library by David Eppstein, which is available at http://www.ics.uci.edu/~eppstein/PADS/ under the MIT license. It has a quadratic runtime and has been described in [Epp2008].

For more information on partial cubes, see the Wikipedia article Partial cube.

5.35.1 Recognition algorithm

Definitions

A **partial cube** is an isometric subgraph G of a *CubeGraph()* (of possibly high dimension). Consequently, the vertices of G can be labelled with binary sequences in such a way that the distance between two vertices $u, v \in G$ is the Hamming distance between their labels.

Tokens and their **action**: in the terminology of [Epp2008], a token represents a transition of the form:

switch the k-th bit of the binary string from 0 to 1

Each token can be matched with a 'reversed' token that performs the same switch in the opposite direction. Alternatively, a token can be seen as a set of disjoint (directed) edges of G, corresponding to the transitions. When a vertex $v \in G$ is the source of such an edge, it is said that the token *acts* on v.

Observations

Shortest paths: in a hypercube, a shortest path between two vertices uses each token at most once. Furthermore, it cannot use both a token and it reverse.

Cycles: a cycle in a partial cube is necessarily even, as hypercubes are bipartite. If an edge e of a cycle C belongs to a token T, then the edge opposite to e in C belongs to the reverse of T.

Incident edges: all $2d_G(v)$ arcs incident to a given vertex belong to as many different tokens.

Algorithm

Labeling: Iteratively, the algorithm selects a vertex $v \in G$, which is naturally associated to $2d(v)$ tokens. It then performs a breadth-first search from v, applying the previous observation on cycles to attribute a token to some of the edges it meets. None of the edges whose token remains undecided after this step can belong to one of those $2d(v)$ tokens, by virtue of the observation on shortest paths.

The labeled edges can then be simplified (contracted) if the previous step did not lead to a contradiction, and the procedure is applied again until the graph is contracted to a single vertex and all edges are labeled.

A partial cube is correctly labeled at this step, but some other graphs can also satisfy the procedure.

Checking the labeling: once all tokens are defined and the vertices are labeled with a binary string, we check that they define an isometric subgraph of the hypercube. To ensure that the distance $d(v_0, u)$ is what we expect for any vertex u, it is sufficient to find, for any vertex u, a neighbor n_u of u whose Hamming distance with v_0 is strictly less than the Hamming distance between u and v_0. Here is the algorithm used to check the labeling:

- For an initial vertex v, run a BFS starting from v, and associate to every other vertex u a token that brings u closer to v. This yields shortest paths from every vertex to v.
- Assuming that the information is computed (and correct) for v, it is easy to update it for a neighbor v' of v. Indeed, if we write T the token that turns v into v', only the vertices which were associated with the reverse of T need to select a new neighbour. All others can remain as they were previously.

  With this second observation, one can efficiently check that the distance between all pairs of vertices are what they should be. In the implementation, the sequence of the sources $(v, v', ...)$ is given by a depth-first search.

5.35.2 Functions

sage.graphs.partial_cube.**breadth_first_level_search**(*G*, *start*)

Generate a sequence of dictionaries, each mapping the vertices at distance `i` from `start` to the set of their neighbours at distance `i+1`.

Originally written by D. Eppstein for the PADS library (http://www.ics.uci.edu/~eppstein/PADS/).

INPUT:

- `G` – a graph to perform the search on.
- `start` – vertex or list of vertices from which to start the traversal.

EXAMPLES:

```
sage: H = digraphs.DeBruijn(3,2)
sage: list(sage.graphs.partial_cube.breadth_first_level_search(H, '00'))
[{'00': {'01', '02'}},
 {'01': {'10', '11', '12'}, '02': {'20', '21', '22'}},
 {'10': set(),
  '11': set(),
  '12': set(),
  '20': set(),
  '21': set(),
  '22': set()}]
```

sage.graphs.partial_cube.**depth_first_traversal**(*G*, *start*)

Generate a sequence of triples (v,w,edgetype) for DFS of graph G.

Originally written by D. Eppstein for the PADS library (http://www.ics.uci.edu/~eppstein/PADS/).

INPUT:

- G – a graph to perform the search on.
- start – vertex or list of vertices from which to start the traversal.

OUTPUT:

- a generator of triples (v,w,edgetype), where edgetype is True if the algorithm is progressing via the edge vw, or False if the algorithm is backtracking via the edge wv.

EXAMPLES:

```
sage: H = digraphs.DeBruijn(3,2)
sage: t = list(sage.graphs.partial_cube.depth_first_traversal(H, '00'))
sage: len(t)
16
```

sage.graphs.partial_cube.**is_partial_cube**(*G*, *certificate=False*)

Test whether the given graph is a partial cube.

A partial cube is a graph that can be isometrically embedded into a hypercube, i.e., its vertices can be labelled with (0,1)-vectors of some fixed length such that the distance between any two vertices in the graph equals the Hamming distance of their labels.

Originally written by D. Eppstein for the PADS library (http://www.ics.uci.edu/~eppstein/PADS/), see also [Epp2008]. The algorithm runs in $O(n^2)$ time, where n is the number of vertices. See the documentation of *partial_cube* for an overview of the algorithm.

INPUT:

- certificate – boolean (default: False); this function returns True or False according to the graph, when certificate = False. When certificate = True and the graph is a partial cube, the function returns (True, mapping), where mapping is an isometric mapping of the vertices of the graph to the vertices of a hypercube ((0, 1)-strings of a fixed length). When certificate = True and the graph is not a partial cube, (False, None) is returned.

EXAMPLES:

The Petersen graph is not a partial cube:

```
sage: g = graphs.PetersenGraph()
sage: g.is_partial_cube()
False
```

All prisms are partial cubes:

```
sage: g = graphs.CycleGraph(10).cartesian_product(graphs.CompleteGraph(2))
sage: g.is_partial_cube()
True
```

5.36 Path Enumeration

This module is meant for all functions related to path enumeration in graphs.

all_paths()	Return the list of all paths between a pair of vertices.
yen_k_shortest_simple_paths()	Return an iterator over the simple paths between a pair of vertices in increasing order of weights.
feng_k_shortest_simple_paths()	Return an iterator over the simple paths between a pair of vertices in increasing order of weights.
all_paths_iterator()	Return an iterator over the paths of `self`.
all_simple_paths()	Return a list of all the simple paths of `self` starting with one of the given vertices.
shortest_simple_paths()	Return an iterator over the simple paths between a pair of vertices.

5.36.1 Functions

sage.graphs.path_enumeration.**all_paths**(*G*, *start*, *end*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return the list of all paths between a pair of vertices.

If `start` is the same vertex as `end`, then `[[start]]` is returned – a list containing the 1-vertex, 0-edge path "`start`".

If `G` has multiple edges, a path will be returned as many times as the product of the multiplicity of the edges along that path depending on the value of the flag `use_multiedges`.

INPUT:

- `start` – a vertex of a graph, where to start
- `end` – a vertex of a graph, where to end
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`
  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

EXAMPLES:

```
sage: eg1 = Graph({0:[1, 2], 1:[4], 2:[3, 4], 4:[5], 5:[6]})
sage: eg1.all_paths(0, 6)
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg2 = graphs.PetersenGraph()
sage: sorted(eg2.all_paths(1, 4))
[[1, 0, 4],
 [1, 0, 5, 7, 2, 3, 4],
 [1, 0, 5, 7, 2, 3, 8, 6, 9, 4],
 [1, 0, 5, 7, 9, 4],
 [1, 0, 5, 7, 9, 6, 8, 3, 4],
 [1, 0, 5, 8, 3, 2, 7, 9, 4],
```

(continues on next page)

(continued from previous page)

```
 [1, 0, 5, 8, 3, 4],
 [1, 0, 5, 8, 6, 9, 4],
 [1, 0, 5, 8, 6, 9, 7, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 8, 5, 0, 4],
 [1, 2, 3, 8, 5, 7, 9, 4],
 [1, 2, 3, 8, 6, 9, 4],
 [1, 2, 3, 8, 6, 9, 7, 5, 0, 4],
 [1, 2, 7, 5, 0, 4],
 [1, 2, 7, 5, 8, 3, 4],
 [1, 2, 7, 5, 8, 6, 9, 4],
 [1, 2, 7, 9, 4],
 [1, 2, 7, 9, 6, 8, 3, 4],
 [1, 2, 7, 9, 6, 8, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 9, 4],
 [1, 6, 8, 3, 4],
 [1, 6, 8, 5, 0, 4],
 [1, 6, 8, 5, 7, 2, 3, 4],
 [1, 6, 8, 5, 7, 9, 4],
 [1, 6, 9, 4],
 [1, 6, 9, 7, 2, 3, 4],
 [1, 6, 9, 7, 2, 3, 8, 5, 0, 4],
 [1, 6, 9, 7, 5, 0, 4],
 [1, 6, 9, 7, 5, 8, 3, 4]]
sage: dg = DiGraph({0:[1, 3], 1:[3], 2:[0, 3]})
sage: sorted(dg.all_paths(0, 3))
[[0, 1, 3], [0, 3]]
sage: ug = dg.to_undirected()
sage: sorted(ug.all_paths(0, 3))
[[0, 1, 3], [0, 2, 3], [0, 3]]

sage: g = Graph([(0, 1), (0, 1), (1, 2), (1, 2)], multiedges=True)
sage: g.all_paths(0, 2, use_multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]

sage: dg = DiGraph({0:[1, 2, 1], 3:[0, 0]}, multiedges=True)
sage: dg.all_paths(3, 1, use_multiedges=True)
[[3, 0, 1], [3, 0, 1], [3, 0, 1], [3, 0, 1]]

sage: g = Graph([(0, 1, 'a'), (0, 1, 'b'), (1, 2, 'c'), (1, 2, 'd')],␣
↪multiedges=True)
sage: g.all_paths(0, 2, use_multiedges=False)
[[0, 1, 2]]
sage: g.all_paths(0, 2, use_multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=True)
[[(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)]]
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=True, labels=True)
[((0, 1, 'b'), (1, 2, 'd')),
 ((0, 1, 'b'), (1, 2, 'c')),
 ((0, 1, 'a'), (1, 2, 'd')),
 ((0, 1, 'a'), (1, 2, 'c'))]
sage: g.all_paths(0, 2, use_multiedges=False, report_edges=True, labels=True)
[((0, 1, 'b'), (1, 2, 'd'))]
sage: g.all_paths(0, 2, use_multiedges=False, report_edges=False, labels=True)
[[0, 1, 2]]
```

(continues on next page)

(continued from previous page)

```
sage: g.all_paths(0, 2, use_multiedges=True, report_edges=False, labels=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

sage.graphs.path_enumeration.**all_paths_iterator**(*self*, *starting_vertices=None*, *ending_vertices=None*, *simple=False*, *max_length=None*, *trivial=False*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return an iterator over the paths of `self`.

The paths are enumerated in increasing length order.

INPUT:

- `starting_vertices` – iterable (default: `None`); vertices from which the paths must start. If `None`, then all vertices of the graph can be starting points.
- `ending_vertices` – iterable (default: `None`); allowed ending vertices of the paths. If `None`, then all vertices are allowed.
- `simple` – boolean (default: `False`); if set to `True`, then only simple paths are considered. Simple paths are paths in which no two arcs share a head or share a tail, i.e. every vertex in the path is entered at most once and exited at most once.
- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`
  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

OUTPUT:

iterator

AUTHOR:

Alexandre Blondin Masse

EXAMPLES:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['d'],␣
↪report_edges=True, simple=True)
sage: list(pi)
```

(continues on next page)

(continued from previous page)

```
[[('a', 'b'), ('b', 'c'), ('c', 'd')]]

sage: g = DiGraph([(0, 1, 'a'), (0, 1, 'b'), (1, 2,'c'), (1, 2,'d')],␣
↪multiedges=True)
sage: pi =  g.all_paths_iterator(starting_vertices=[0], use_multiedges=True)
sage: for _ in range(6):
....:     print(next(pi))
[0, 1]
[0, 1]
[0, 1, 2]
[0, 1, 2]
[0, 1, 2]
[0, 1, 2]
sage: pi =  g.all_paths_iterator(starting_vertices=[0], use_multiedges=True,␣
↪report_edges=True, labels=True)
sage: for _ in range(6):
....:     print(next(pi))
[(0, 1, 'b')]
[(0, 1, 'a')]
[(0, 1, 'b'), (1, 2, 'd')]
[(0, 1, 'b'), (1, 2, 'c')]
[(0, 1, 'a'), (1, 2, 'd')]
[(0, 1, 'a'), (1, 2, 'c')]
sage: list(g.all_paths_iterator(starting_vertices=[0, 1], ending_vertices=[2],␣
↪use_multiedges=False, report_edges=True, labels=True, simple=True))
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
sage: list(g.all_paths_iterator(starting_vertices=[0, 1], ending_vertices=[2],␣
↪use_multiedges=False, report_edges=False, labels=True))
[[1, 2], [0, 1, 2]]
sage: list(g.all_paths_iterator(use_multiedges=True, report_edges=False,␣
↪labels=True, max_length=1))
[[1, 2], [1, 2], [0, 1], [0, 1]]
sage: list(g.all_paths_iterator(use_multiedges=True, report_edges=True,␣
↪labels=True, max_length=1))
[[(1, 2, 'd')], [(1, 2, 'c')], [(0, 1, 'b')], [(0, 1, 'a')]]

sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: pi = g.all_paths_iterator()
sage: for _ in range(7):    # py2
....:     print(next(pi))   # py2
['d', 'c']
['b', 'c']
['c', 'd']
['a', 'a']
['a', 'b']
['a', 'a', 'a']
['a', 'a', 'b']
sage: pi = g.all_paths_iterator()
sage: [len(next(pi)) - 1 for _ in range(7)]
[1, 1, 1, 1, 1, 2, 2]
```

It is possible to precise the allowed starting and/or ending vertices:

```
sage: pi = g.all_paths_iterator(starting_vertices=['a'])
sage: for _ in range(5):    # py2
....:     print(next(pi))   # py2
```

(continues on next page)

(continued from previous page)

```
['a', 'a']
['a', 'b']
['a', 'a', 'a']
['a', 'a', 'b']
['a', 'b', 'c']
sage: pi = g.all_paths_iterator(starting_vertices=['a'])
sage: [len(next(pi)) - 1 for _ in range(5)]
[1, 1, 2, 2, 2]
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b'])
sage: for _ in range(5):
....:     print(next(pi))
['a', 'b']
['a', 'a', 'b']
['a', 'a', 'a', 'b']
['a', 'a', 'a', 'a', 'b']
['a', 'a', 'a', 'a', 'a', 'b']
```

One may prefer to enumerate only simple paths (see *all_simple_paths()*):

```
sage: pi = g.all_paths_iterator(simple=True)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'c'],
    ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'c'], ['d', 'c', 'd'],
    ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True)
sage: [len(p) - 1 for p in pi]
[1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
```

Or simply bound the length of the enumerated paths:

```
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b', 'c
↪'], max_length=6)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'b'], ['a', 'a', 'b'], ['a', 'b', 'c'], ['a', 'a', 'a', 'b'],
    ['a', 'a', 'b', 'c'], ['a', 'a', 'a', 'a', 'b'],
    ['a', 'a', 'a', 'b', 'c'], ['a', 'b', 'c', 'd', 'c'],
    ['a', 'a', 'a', 'a', 'a', 'b'], ['a', 'a', 'a', 'a', 'b', 'c'],
    ['a', 'a', 'b', 'c', 'd', 'c'],
    ['a', 'a', 'a', 'a', 'a', 'a', 'b'],
    ['a', 'a', 'a', 'a', 'a', 'b', 'c'],
    ['a', 'a', 'a', 'b', 'c', 'd', 'c'],
    ['a', 'b', 'c', 'd', 'c', 'd', 'c']]
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b', 'c
↪'], max_length=6)
sage: [len(p) - 1 for p in pi]
[1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6]
```

By default, empty paths are not enumerated, but it may be parametrized:

```
sage: pi = g.all_paths_iterator(simple=True, trivial=True)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a'], ['b'], ['c'], ['d'], ['a', 'a'], ['a', 'b'], ['b', 'c'],
    ['c', 'd'], ['d', 'c'], ['a', 'b', 'c'], ['b', 'c', 'd'],
    ['c', 'd', 'c'], ['d', 'c', 'd'], ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True, trivial=True)
sage: [len(p) - 1 for p in pi]
[0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
```

(continues on next page)

(continued from previous page)

```
sage: pi = g.all_paths_iterator(simple=True, trivial=False)
sage: sorted(list(pi), key=lambda x:(len(x), x))
[['a', 'a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'c'],
    ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'c'], ['d', 'c', 'd'],
    ['a', 'b', 'c', 'd']]
sage: pi = g.all_paths_iterator(simple=True, trivial=False)
sage: [len(p) - 1 for p in pi]
[1, 1, 1, 1, 1, 2, 2, 2, 2, 3]
```

sage.graphs.path_enumeration.**all_simple_paths**(*self*, *starting_vertices=None*, *ending_vertices=None*, *max_length=None*, *trivial=False*, *use_multiedges=False*, *report_edges=False*, *labels=False*)

Return a list of all the simple paths of `self` starting with one of the given vertices.

Simple paths are paths in which no two arcs share a head or share a tail, i.e. every vertex in the path is entered at most once and exited at most once.

INPUT:

- `starting_vertices` – list (default: `None`); vertices from which the paths must start. If `None`, then all vertices of the graph can be starting points.
- `ending_vertices` – iterable (default: `None`); allowed ending vertices of the paths. If `None`, then all vertices are allowed.
- `max_length` – non negative integer (default: `None`); the maximum length of the enumerated paths. If set to `None`, then all lengths are allowed.
- `trivial` - boolean (default: `False`); if set to `True`, then the empty paths are also enumerated.
- `use_multiedges` – boolean (default: `False`); this parameter is used only if the graph has multiple edges.
  - If `False`, the graph is considered as simple and an edge label is arbitrarily selected for each edge as in *sage.graphs.generic_graph.GenericGraph.to_simple()* if `report_edges` is `True`
  - If `True`, a path will be reported as many times as the edges multiplicities along that path (when `report_edges = False` or `labels = False`), or with all possible combinations of edge labels (when `report_edges = True` and `labels = True`)
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.

OUTPUT:

list

Note: Although the number of simple paths of a finite graph is always finite, computing all its paths may take a very long time.

EXAMPLES:

```
sage: g = DiGraph({0: [0, 1], 1: [2], 2: [3], 3: [2]}, loops=True)
sage: g.all_simple_paths()
[[3, 2],
 [2, 3],
 [1, 2],
 [0, 0],
 [0, 1],
 [0, 1, 2],
 [1, 2, 3],
 [2, 3, 2],
 [3, 2, 3],
 [0, 1, 2, 3]]

sage: g = DiGraph([(0, 1, 'a'), (0, 1, 'b'), (1, 2,'c'), (1, 2,'d')],␣
↪multiedges=True)
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=False)
[[0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True)
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True, report_edges=True)
[[(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)], [(0, 1), (1, 2)]]
sage: g.all_simple_paths(starting_vertices=[0], ending_vertices=[2], use_
↪multiedges=True, report_edges=True, labels=True)
[[(0, 1, 'b'), (1, 2, 'd')],
 [(0, 1, 'b'), (1, 2, 'c')],
 [(0, 1, 'a'), (1, 2, 'd')],
 [(0, 1, 'a'), (1, 2, 'c')]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=True, labels=True)
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=False, labels=True)
[[1, 2], [0, 1, 2]]
sage: g.all_simple_paths(use_multiedges=True, report_edges=False, labels=True)
[[1, 2], [1, 2], [0, 1], [0, 1], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
sage: g.all_simple_paths(starting_vertices=[0, 1], ending_vertices=[2], use_
↪multiedges=False, report_edges=True, labels=True, trivial=True)
[[(1, 2, 'd')], [(0, 1, 'b'), (1, 2, 'd')]]
```

One may compute all paths having specific starting and/or ending vertices:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: g.all_simple_paths(starting_vertices=['a'])
[['a', 'a'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'd']]
sage: g.all_simple_paths(starting_vertices=['a'], ending_vertices=['c'])
[['a', 'b', 'c']]
sage: g.all_simple_paths(starting_vertices=['a'], ending_vertices=['b', 'c'])
[['a', 'b'], ['a', 'b', 'c']]
```

It is also possible to bound the length of the paths:

```
sage: g = DiGraph({0: [0, 1], 1: [2], 2: [3], 3: [2]}, loops=True)
sage: g.all_simple_paths(max_length=2)
```

(continues on next page)

(continued from previous page)

```
[[3, 2],
 [2, 3],
 [1, 2],
 [0, 0],
 [0, 1],
 [0, 1, 2],
 [1, 2, 3],
 [2, 3, 2],
 [3, 2, 3]]
```

By default, empty paths are not enumerated, but this can be parametrized:

```
sage: g = DiGraph({'a': ['a', 'b'], 'b': ['c'], 'c': ['d'], 'd': ['c']},␣
↪loops=True)
sage: g.all_simple_paths(starting_vertices=['a'], trivial=True)
[['a'], ['a', 'a'], ['a', 'b'], ['a', 'b', 'c'],
 ['a', 'b', 'c', 'd']]
sage: g.all_simple_paths(starting_vertices=['a'], trivial=False)
[['a', 'a'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'd']]
```

sage.graphs.path_enumeration.**feng_k_shortest_simple_paths**(*self*, *source*, *target*, *weight_function=None*, *by_weight=False*, *report_edges=False*, *labels=False*, *report_weight=False*)

Return an iterator over the simple paths between a pair of vertices in increasing order of weights.

Works only for directed graphs.

For unweighted graphs, paths are returned in order of increasing number of edges.

In case of weighted graphs, negative weights are not allowed.

If `source` is the same vertex as `target`, then `[[source]]` is returned – a list containing the 1-vertex, 0-edge path `source`.

The loops and the multiedges if present in the given graph are ignored and only minimum of the edge labels is kept in case of multiedges.

INPUT:

- `source` – a vertex of the graph, where to start
- `target` – a vertex of the graph, where to end
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.
- `report_weight` – boolean (default: `False`); if `False`, just the path between `source` and `target` is returned. Otherwise a tuple of path length and path is returned.

ALGORITHM:

This algorithm can be divided into two parts. Firstly, it determines the shortest path from `source` to `target`. Then, it determines all the other k-shortest paths. This algorithm finds the deviations of previous shortest paths to determine the next shortest paths. This algorithm finds the candidate paths more efficiently using a node classification technique. At first the candidate path is separated by its deviation node as prefix and suffix. Then the algorithm classify the nodes as red, yellow and green. A node on the prefix is assigned a red color, a node that can reach t (the destination node) through a shortest path without visiting a red node is assigned a green color, and all other nodes are assigned a yellow color. When searching for the suffix of a candidate path, all green nodes are bypassed, and `Dijkstra's algorithm` is applied to find an all-yellow-node subpath. Since on average the number of yellow nodes is much smaller than n, this algorithm has a much lower average-case running time.

Time complexity is $O(kn(m + n \log n))$ where n is the number of vertices and m is the number of edges and k is the number of shortest paths needed to find. Its average running time is much smaller as compared to $Yen's$ algorithm.

See [Feng2014] for more details on this algorithm.

EXAMPLES:

```
sage: from sage.graphs.path_enumeration import feng_k_shortest_simple_paths
sage: g = DiGraph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,
↪ 5, 30)])
sage: list(feng_k_shortest_simple_paths(g, 1, 5, by_weight=True))
[[1, 3, 5], [1, 2, 5], [1, 4, 5]]
sage: list(feng_k_shortest_simple_paths(g, 1, 5))
[[1, 4, 5], [1, 3, 5], [1, 2, 5]]
sage: list(feng_k_shortest_simple_paths(g, 1, 1))
[[1]]
sage: list(feng_k_shortest_simple_paths(g, 1, 5, report_edges=True, labels=True))
[[(1, 4, 30), (4, 5, 30)], [(1, 3, 10), (3, 5, 10)], [(1, 2, 20), (2, 5, 20)]]
sage: list(feng_k_shortest_simple_paths(g, 1, 5, report_edges=True, labels=True,␣
↪by_weight=True))
[[(1, 3, 10), (3, 5, 10)], [(1, 2, 20), (2, 5, 20)], [(1, 4, 30), (4, 5, 30)]]
sage: list(feng_k_shortest_simple_paths(g, 1, 5, report_edges=True, labels=True,␣
↪by_weight=True, report_weight=True))
[(20, [(1, 3, 10), (3, 5, 10)]),
 (40, [(1, 2, 20), (2, 5, 20)]),
 (60, [(1, 4, 30), (4, 5, 30)])]

sage: from sage.graphs.path_enumeration import feng_k_shortest_simple_paths
sage: g = DiGraph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,
↪ 5, 30), (1, 6, 100), (5, 6, 5)])
sage: list(feng_k_shortest_simple_paths(g, 1, 6, by_weight = True))
[[1, 3, 5, 6], [1, 2, 5, 6], [1, 4, 5, 6], [1, 6]]
sage: list(feng_k_shortest_simple_paths(g, 1, 6))
[[1, 6], [1, 4, 5, 6], [1, 3, 5, 6], [1, 2, 5, 6]]
sage: list(feng_k_shortest_simple_paths(g, 1, 6, report_edges=True, labels=True,␣
↪by_weight=True, report_weight=True))
[(25, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (45, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (65, [(1, 4, 30), (4, 5, 30), (5, 6, 5)]),
 (100, [(1, 6, 100)])]
sage: list(feng_k_shortest_simple_paths(g, 1, 6, report_edges=True, labels=True,␣
↪report_weight=True))
[(1, [(1, 6, 100)]),
 (3, [(1, 4, 30), (4, 5, 30), (5, 6, 5)]),
```

(continues on next page)

(continued from previous page)

```
 (3, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (3, [(1, 2, 20), (2, 5, 20), (5, 6, 5)])]
sage: from sage.graphs.path_enumeration import feng_k_shortest_simple_paths
sage: g = DiGraph([(1, 2, 5), (2, 3, 0), (1, 4, 2), (4, 5, 1), (5, 3, 0)])
sage: list(feng_k_shortest_simple_paths(g, 1, 3, by_weight=True))
[[1, 4, 5, 3], [1, 2, 3]]
sage: list(feng_k_shortest_simple_paths(g, 1, 3))
[[1, 2, 3], [1, 4, 5, 3]]
sage: list(feng_k_shortest_simple_paths(g, 1, 3, report_weight=True))
[(2, [1, 2, 3]), (3, [1, 4, 5, 3])]
sage: list(feng_k_shortest_simple_paths(g, 1, 3, report_weight=True, report_
↪edges=True))
[(2, [(1, 2), (2, 3)]), (3, [(1, 4), (4, 5), (5, 3)])]
sage: list(feng_k_shortest_simple_paths(g, 1, 3, report_weight=True, report_
↪edges=True, by_weight=True))
[(3, [(1, 4), (4, 5), (5, 3)]), (5, [(1, 2), (2, 3)])]
sage: list(feng_k_shortest_simple_paths(g, 1, 3, report_weight=True, report_
↪edges=True, by_weight=True, labels=True))
[(3, [(1, 4, 2), (4, 5, 1), (5, 3, 0)]), (5, [(1, 2, 5), (2, 3, 0)])]
```

sage.graphs.path_enumeration.**shortest_simple_paths**(*self*, *source*, *target*, *weight_function=None*, *by_weight=False*, *algorithm=None*, *report_edges=False*, *labels=False*, *report_weight=False*)

Return an iterator over the simple paths between a pair of vertices.

This method returns an iterator over the simple paths (i.e., without repetition) from `source` to `target`. By default (`by_weight` is `False`), the paths are reported by increasing number of edges. When `by_weight` is `True`, the paths are reported by increasing weights.

In case of weighted graphs negative weights are not allowed.

If `source` is the same vertex as `target`, then `[[source]]` is returned – a list containing the 1-vertex, 0-edge path `source`.

By default `Yen's` algorithm [Yen1970] is used for undirected graphs and `Feng's` algorithm is used for directed graphs [Feng2014].

The loops and the multiedges if present in the given graph are ignored and only minimum of the edge labels is kept in case of multiedges.

INPUT:

- `source` – a vertex of the graph, where to start
- `target` – a vertex of the graph, where to end
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `algorithm` – string (default: `None`); the algorithm to use in computing `k` shortest paths of `self`. The following algorithms are supported:
  - `"Yen"` – Yen's algorithm [Yen1970]

  - `"Feng"` – an improved version of Yen's algorithm but that works only for directed graphs [Feng2014]

- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges. When set to `False`, the `labels` parameter is ignored.
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.
- `report_weight` – boolean (default: `False`); if `False`, just the path between `source` and `target` is returned. Otherwise a tuple of path length and path is returned.

EXAMPLES:

```
sage: g = DiGraph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,
↪ 5, 30)])
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Yen"))
[[1, 3, 5], [1, 2, 5], [1, 4, 5]]
sage: list(g.shortest_simple_paths(1, 5, algorithm="Yen"))
[[1, 2, 5], [1, 3, 5], [1, 4, 5]]
sage: list(g.shortest_simple_paths(1, 1))
[[1]]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, report_edges=True,␣
↪report_weight=True, labels=True))
[(20, [(1, 3, 10), (3, 5, 10)]),
 (40, [(1, 2, 20), (2, 5, 20)]),
 (60, [(1, 4, 30), (4, 5, 30)])]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Feng", report_
↪edges=True, report_weight=True))
[(20, [(1, 3), (3, 5)]), (40, [(1, 2), (2, 5)]), (60, [(1, 4), (4, 5)])]
sage: list(g.shortest_simple_paths(1, 5, report_edges=True, report_weight=True))
[(2, [(1, 4), (4, 5)]), (2, [(1, 3), (3, 5)]), (2, [(1, 2), (2, 5)])]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, report_edges=True))
[[(1, 3), (3, 5)], [(1, 2), (2, 5)], [(1, 4), (4, 5)]]
sage: list(g.shortest_simple_paths(1, 5, by_weight=True, algorithm="Feng", report_
↪edges=True, labels=True))
[[(1, 3, 10), (3, 5, 10)], [(1, 2, 20), (2, 5, 20)], [(1, 4, 30), (4, 5, 30)]]
sage: g = Graph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,␣
↪5, 30), (1, 6, 100), (5, 6, 5)])
sage: list(g.shortest_simple_paths(1, 6, by_weight = True))
[[1, 3, 5, 6], [1, 2, 5, 6], [1, 4, 5, 6], [1, 6]]
sage: list(g.shortest_simple_paths(1, 6, algorithm="Yen"))
[[1, 6], [1, 2, 5, 6], [1, 3, 5, 6], [1, 4, 5, 6]]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, report_weight=True,␣
↪labels=True))
[(1, [(1, 6, 100)]),
 (3, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (3, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (3, [(1, 4, 30), (4, 5, 30), (5, 6, 5)])]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, report_weight=True,␣
↪labels=True, by_weight=True))
[(25, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (45, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (65, [(1, 4, 30), (4, 5, 30), (5, 6, 5)]),
 (100, [(1, 6, 100)])]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, labels=True, by_
↪weight=True))
[[(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)],
```

(continues on next page)

(continued from previous page)

```
 [(1, 6, 100)]]
sage: list(g.shortest_simple_paths(1, 6, report_edges=True, labels=True))
[[(1, 6, 100)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)]]
```

sage.graphs.path_enumeration.**yen_k_shortest_simple_paths**(*self*, *source*, *target*, *weight_function=None*, *by_weight=False*, *report_edges=False*, *labels=False*, *report_weight=False*)

Return an iterator over the simple paths between a pair of vertices in increasing order of weights.

For unweighted graphs paths are returned in order of increasing number of edges.

In case of weighted graphs negative weights are not allowed.

If `source` is the same vertex as `target`, then `[[source]]` is returned – a list containing the 1-vertex, 0-edge path `source`.

The loops and the multiedges if present in the given graph are ignored and only minimum of the edge labels is kept in case of multiedges.

INPUT:

- `source` – a vertex of the graph, where to start
- `target` – a vertex of the graph, where to end
- `weight_function` – function (default: `None`); a function that takes as input an edge `(u, v, l)` and outputs its weight. If not `None`, `by_weight` is automatically set to `True`. If `None` and `by_weight` is `True`, we use the edge label `l` as a weight.
- `by_weight` – boolean (default: `False`); if `True`, the edges in the graph are weighted, otherwise all edges have weight 1
- `report_edges` – boolean (default: `False`); whether to report paths as list of vertices (default) or list of edges, if `False` then `labels` parameter is ignored
- `labels` – boolean (default: `False`); if `False`, each edge is simply a pair `(u, v)` of vertices. Otherwise a list of edges along with its edge labels are used to represent the path.
- `report_weight` – boolean (default: `False`); if `False`, just the path between `source` and `target` is returned. Otherwise a tuple of path length and path is returned.

ALGORITHM:

This algorithm can be divided into two parts. Firstly, it determines a shortest path from `source` to `target`. Then, it determines all the other k-shortest paths. This algorithm finds the deviations of previous shortest paths to determine the next shortest paths.

Time complexity is $O(kn(m + n \log n))$ where n is the number of vertices and m is the number of edges and k is the number of shortest paths needed to find.

See [Yen1970] and the Wikipedia article Yen's_algorithm for more details on the algorithm.

EXAMPLES:

```
sage: from sage.graphs.path_enumeration import yen_k_shortest_simple_paths
sage: g = DiGraph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,
↪ 5, 30)])
sage: list(yen_k_shortest_simple_paths(g, 1, 5, by_weight=True))
[[1, 3, 5], [1, 2, 5], [1, 4, 5]]
sage: list(yen_k_shortest_simple_paths(g, 1, 5))
[[1, 2, 5], [1, 3, 5], [1, 4, 5]]
sage: list(yen_k_shortest_simple_paths(g, 1, 1))
[[1]]
sage: list(yen_k_shortest_simple_paths(g, 1, 5, by_weight=True, report_edges=True,
↪ report_weight=True, labels=True))
[(20, [(1, 3, 10), (3, 5, 10)]),
 (40, [(1, 2, 20), (2, 5, 20)]),
 (60, [(1, 4, 30), (4, 5, 30)])]
sage: list(yen_k_shortest_simple_paths(g, 1, 5, by_weight=True, report_edges=True,
↪ report_weight=True))
[(20, [(1, 3), (3, 5)]), (40, [(1, 2), (2, 5)]), (60, [(1, 4), (4, 5)])]
sage: list(yen_k_shortest_simple_paths(g, 1, 5, report_edges=True, report_
↪weight=True))
[(2, [(1, 2), (2, 5)]), (2, [(1, 3), (3, 5)]), (2, [(1, 4), (4, 5)])]
sage: list(yen_k_shortest_simple_paths(g, 1, 5, by_weight=True, report_
↪edges=True))
[[(1, 3), (3, 5)], [(1, 2), (2, 5)], [(1, 4), (4, 5)]]
sage: list(yen_k_shortest_simple_paths(g, 1, 5, by_weight=True, report_edges=True,
↪ labels=True))
[[(1, 3, 10), (3, 5, 10)], [(1, 2, 20), (2, 5, 20)], [(1, 4, 30), (4, 5, 30)]]
sage: from sage.graphs.path_enumeration import yen_k_shortest_simple_paths
sage: g = Graph([(1, 2, 20), (1, 3, 10), (1, 4, 30), (2, 5, 20), (3, 5, 10), (4,␣
↪5, 30), (1, 6, 100), (5, 6, 5)])
sage: list(yen_k_shortest_simple_paths(g, 1, 6, by_weight = True))
[[1, 3, 5, 6], [1, 2, 5, 6], [1, 4, 5, 6], [1, 6]]
sage: list(yen_k_shortest_simple_paths(g, 1, 6))
[[1, 6], [1, 2, 5, 6], [1, 3, 5, 6], [1, 4, 5, 6]]
sage: list(yen_k_shortest_simple_paths(g, 1, 6, report_edges=True, report_
↪weight=True, labels=True))
[(1, [(1, 6, 100)]),
 (3, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (3, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (3, [(1, 4, 30), (4, 5, 30), (5, 6, 5)])]
sage: list(yen_k_shortest_simple_paths(g, 1, 6, report_edges=True, report_
↪weight=True, labels=True, by_weight=True))
[(25, [(1, 3, 10), (3, 5, 10), (5, 6, 5)]),
 (45, [(1, 2, 20), (2, 5, 20), (5, 6, 5)]),
 (65, [(1, 4, 30), (4, 5, 30), (5, 6, 5)]),
 (100, [(1, 6, 100)])]
sage: list(yen_k_shortest_simple_paths(g, 1, 6, report_edges=True, labels=True,␣
↪by_weight=True))
[[(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)],
 [(1, 6, 100)]]
sage: list(yen_k_shortest_simple_paths(g, 1, 6, report_edges=True, labels=True))
[[(1, 6, 100)],
 [(1, 2, 20), (2, 5, 20), (5, 6, 5)],
 [(1, 3, 10), (3, 5, 10), (5, 6, 5)],
 [(1, 4, 30), (4, 5, 30), (5, 6, 5)]]
```

5.37 GenericGraph Cython functions

AUTHORS:

- Robert L. Miller (2007-02-13): initial version
- Robert W. Bradshaw (2007-03-31): fast spring layout algorithms
- Nathann Cohen : exhaustive search

class sage.graphs.generic_graph_pyx.**GenericGraph_pyx**
Bases: sage.structure.sage_object.SageObject

class sage.graphs.generic_graph_pyx.**SubgraphSearch**
Bases: object

This class implements methods to exhaustively search for copies of a graph H in a larger graph G.

It is possible to look for induced subgraphs instead, and to iterate or count the number of their occurrences.

ALGORITHM:

The algorithm is a brute-force search. Let $V(H) = \{h_1, \ldots, h_k\}$. It first tries to find in G a possible representative of h_1, then a representative of h_2 compatible with h_1, then a representative of h_3 compatible with the first two, etc.

This way, most of the time we need to test far less than $k!\binom{|V(G)|}{k}$ subsets, and hope this brute-force technique can sometimes be useful.

Note: This algorithm does not take vertex/edge labels into account.

---

cardinality()
Returns the number of labelled subgraphs of G isomorphic to H.

Note: This method counts the subgraphs by enumerating them all ! Hence it probably is not a good idea to count their number before enumerating them :-)

EXAMPLES:

Counting the number of labelled P_3 in P_5:

```
sage: from sage.graphs.generic_graph_pyx import SubgraphSearch
sage: g = graphs.PathGraph(5)
sage: h = graphs.PathGraph(3)
sage: S = SubgraphSearch(g, h)
sage: S.cardinality()
6
```

sage.graphs.generic_graph_pyx.**binary_string_from_dig6**(*s*, *n*)
A helper function for the dig6 format.

INPUT:

- s – a graph6 string
- n – the length of the binary string encoded by s.

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import binary_string_from_dig6
sage: binary_string_from_dig6('?????_@?CG??B??@OG?C?G???GO??W@a???CO???OACC?OA?
↪P@G??O??????G??C????c?G?CC?_?@???C_??_?C????PO?C_??AA?OOAHCA___?CC?A?CAOGO??????
↪A??G?GR?C?_o`???g???A_C?OG??O?G_IA????_QO@EG???O??C?_?C@?G???@?_??AC?AO?a???O???
↪??A?_Dw?H???__O@AAOAACd?_C??G?G@??GO?_???O@?_O??W??@P???AG??B?????G??GG???A??@?
↪aC_G@A??O??_?A?????O@Z?_@M????GQ@_G@?C?', 63)

↪'00000000000000000000000000000010000000001000000000100001000000000000000000011000000000000000
↪..
↪10000000000000000000000000000000100000000010111011000000100000000001001110000000000000000000000
↪'
sage: binary_string_from_dig6('???C?@AA?_?A?O?C??S??O?q_?P?CHD??@?C?GC???C??GG?C_?
↪?O?COG????I?J??Q??O?_@@??@??????', 32)

↪'00000000000000000000010000000000000010000100000100000001000000000000000010000000100000.
↪..
↪01000000000000010001000000100000000000000000000000000000000101000000000101100000000000010010000000
↪'
```

sage.graphs.generic_graph_pyx.**binary_string_from_graph6**(*s*, *n*)

Decodes a binary string from its graph6 representation

This helper function is the inverse of R from [McK2015].

INPUT:

- `s` – a graph6 string
- `n` – the length of the binary string encoded by `s`.

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import binary_string_from_graph6
sage: binary_string_from_graph6('?????_@?CG??B??@OG?C?G???GO??W@a???CO???OACC?OA?
↪P@G??O??????G??C????c?G?CC?_?@???C_??_?C????PO?C_??AA?OOAHCA___?CC?A?CAOGO??????
↪A??G?GR?C?_o`???g???A_C?OG??O?G_IA????_QO@EG???O??C?_?C@?G???@?_??AC?AO?a???O???
↪??A?_Dw?H???__O@AAOAACd?_C??G?G@??GO?_???O@?_O??W??@P???AG??B?????G??GG???A??@?
↪aC_G@A??O??_?A?????O@Z?_@M????GQ@_G@?C?', 63)

↪'00000000000000000000000000000010000000001000000000100001000000000000000000011000000000000000
↪..
↪10000000000000000000000000000000100000000010111011000000100000000001001110000000000000000000000
↪'
sage: binary_string_from_graph6('???C?@AA?_?A?O?C??S??O?q_?P?CHD??@?C?GC???C??GG?
↪C_??O?COG????I?J??Q??O?_@@??@??????', 32)

↪'00000000000000000000010000000000000010000100000100000001000000000000000010000000100000.
↪..
↪01000000000000010001000000100000000000000000000000000000000101000000000101100000000000010010000000
↪'
```

sage.graphs.generic_graph_pyx.**binary_string_to_graph6**(*x*)

Transforms a binary string into its graph6 representation.

This helper function is named R in [McK2015].

INPUT:

- `x` – a binary string.

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import binary_string_to_graph6
sage: binary_string_to_graph6('1101110101101100101110000011000000010000000001')
'vUqwK@?G'
```

sage.graphs.generic_graph_pyx.**find_hamiltonian**(*G*, *max_iter=100000*, *reset_bound=30000*, *backtrack_bound=1000*, *find_path=False*)

Randomized backtracking for finding Hamiltonian cycles and paths.

ALGORITHM:

A path `P` is maintained during the execution of the algorithm. Initially the path will contain an edge of the graph. Every 10 iterations the path is reversed. Every `reset_bound` iterations the path will be cleared and the procedure is restarted. Every `backtrack_bound` steps we discard the last five vertices and continue with the procedure. The total number of steps in the algorithm is controlled by `max_iter`. If a Hamiltonian cycle or Hamiltonian path is found it is returned. If the number of steps reaches `max_iter` then a longest path is returned. See OUTPUT for more details.

INPUT:

- `G` – graph
- `max_iter` – maximum number of iterations
- **`reset_bound` – number of iterations before restarting the** procedure
- **`backtrack_bound` – number of iterations to elapse before** discarding the last 5 vertices of the path.
- **`find_path` – (default: `False`) if set to `True`, will** search a Hamiltonian path; if `False`, will search for a Hamiltonian cycle

OUTPUT:

A pair `(B, P)`, where `B` is a Boolean and `P` is a list of vertices.

- If `B` is `True` and `find_path` is `False`, `P` represents a Hamiltonian cycle.
- If `B` is `True` and `find_path` is `True`, `P` represents a Hamiltonian path.
- If `B` is `False`, then `P` represents the longest path found during the execution of the algorithm.

Warning: May loop endlessly when run on a graph with vertices of degree 1.

EXAMPLES:

First we try the algorithm in the Dodecahedral graph, which is Hamiltonian, so we are able to find a Hamiltonian cycle and a Hamiltonian path:

```
sage: from sage.graphs.generic_graph_pyx import find_hamiltonian as fh
sage: G=graphs.DodecahedralGraph()
sage: fh(G)
(True, [12, 11, 10, 9, 13, 14, 15, 5, 4, 3, 2, 6, 7, 8, 1, 0, 19, 18, 17, 16])
sage: fh(G,find_path=True)
(True, [10, 0, 19, 3, 4, 5, 15, 16, 17, 18, 11, 12, 13, 9, 8, 1, 2, 6, 7, 14])
```

Another test, now in the Möbius-Kantor graph which is also Hamiltonian, as in our previous example, we are able to find a Hamiltonian cycle and path:

```
sage: G=graphs.MoebiusKantorGraph()
sage: fh(G)
(True, [15, 10, 2, 3, 4, 5, 13, 8, 11, 14, 6, 7, 0, 1, 9, 12])
sage: fh(G,find_path=True)
(True, [10, 15, 7, 6, 5, 4, 12, 9, 14, 11, 3, 2, 1, 0, 8, 13])
```

Now, we try the algorithm on a non Hamiltonian graph, the Petersen graph. This graph is known to be hypohamiltonian, so a Hamiltonian path can be found:

```
sage: G=graphs.PetersenGraph()
sage: fh(G)
(False, [9, 4, 0, 1, 6, 8, 5, 7, 2, 3])
sage: fh(G,find_path=True)
(True, [7, 2, 1, 0, 5, 8, 6, 9, 4, 3])
```

We now show the algorithm working on another known hypohamiltonian graph, the generalized Petersen graph with parameters 11 and 2:

```
sage: G=graphs.GeneralizedPetersenGraph(11,2)
sage: fh(G)
(False, [7, 8, 9, 10, 0, 1, 2, 3, 14, 12, 21, 19, 17, 6, 5, 4, 15, 13, 11, 20, 18,
↪ 16])
sage: fh(G,find_path=True)
(True, [2, 1, 12, 21, 10, 0, 11, 13, 15, 17, 19, 8, 7, 6, 5, 4, 3, 14, 16, 18, 20,
↪ 9])
```

Finally, an example on a graph which does not have a Hamiltonian path:

```
sage: G=graphs.HyperStarGraph(5,2)
sage: fh(G,find_path=False)
(False, ['00110', '10100', '01100', '11000', '01010', '10010', '00011', '10001',
↪'00101'])
sage: fh(G,find_path=True)
(False, ['01001', '10001', '00101', '10100', '00110', '10010', '01010', '11000',
↪'01100'])
```

sage.graphs.generic_graph_pyx.**int_to_binary_string**(*n*)

A quick python int to binary string conversion.

INPUT:

- n (integer)

EXAMPLES:

```
sage: sage.graphs.generic_graph_pyx.int_to_binary_string(389)
'110000101'
sage: Integer(389).binary()
'110000101'
sage: sage.graphs.generic_graph_pyx.int_to_binary_string(2007)
'11111010111'
```

sage.graphs.generic_graph_pyx.**layout_split**(*layout_function*, *G*, ***options*)

Graph each component of G separately with layout_function, placing them adjacent to each other.

This is done because several layout methods need the input graph to be connected. For instance, on a disconnected graph, the spring layout will push components further and further from each other without bound, resulting in very tight clumps for each component.

Note: If the axis are scaled to fit the plot in a square, the horizontal distance may end up being "squished" due to the several adjacent components.

EXAMPLES:

```
sage: G = graphs.DodecahedralGraph()
sage: for i in range(10): G.add_cycle(list(range(100*i, 100*i+3)))
sage: from sage.graphs.generic_graph_pyx import layout_split, spring_layout_fast
sage: D = layout_split(spring_layout_fast, G); D  # random
{0: [0.77..., 0.06...],
 ...
 902: [3.13..., 0.22...]}
```

AUTHOR:

Robert Bradshaw

sage.graphs.generic_graph_pyx.**length_and_string_from_graph6**(*s*)

Returns a pair `(length,graph6_string)` from a graph6 string of unknown length.

This helper function is the inverse of N from [McK2015].

INPUT:

- `s` – a graph6 string describing an binary vector (and encoding its length).

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import length_and_string_from_graph6
sage: length_and_string_from_graph6('~??~?????_@?CG??B??@OG?C?G???GO??W@a???CO???
↪OACC?OA?P@G??O??????G??C????c?G?CC?_?@???C_??_?C????PO?C_??AA?OOAHCA____?CC?A?
↪CAOGO??????A??G?GR?C?_o`???g???A_C?OG??O?G_IA????_QO@EG???O??C?_?C@?G???@?_??AC?
↪AO?a???O?????A?_Dw?H???__O@AAOAACd?_C??G?G@??GO?_???O@?_O??W??@P???AG??B?????G??
↪GG???A??@?aC_G@A??O??_?A?????O@Z?_@M????GQ@_G@?C?')
(63, '?????_@?CG??B??@OG?C?G???GO??W@a???CO???OACC?OA?P@G??O??????G??C????c?G?CC?_
↪?@???C_??_?C????PO?C_??AA?OOAHCA____?CC?A?CAOGO??????A??G?GR?C?_o`???g???A_C?OG??
↪O?G_IA????_QO@EG???O??C?_?C@?G???@?_??AC?AO?a???O?????A?_Dw?H???__O@AAOAACd?_C??
↪G?G@??GO?_???O@?_O??W??@P???AG??B?????G??GG???A??@?aC_G@A??O??_?A?????O@Z?_@M???
↪?GQ@_G@?C?')
sage: length_and_string_from_graph6('_???C?@AA?_?A?O?C??S??O?q_?P?CHD??@?C?GC???C?
↪?GG?C_??O?COG????I?J??Q??O?_@@??@??????')
(32, '???C?@AA?_?A?O?C??S??O?q_?P?CHD??@?C?GC???C??GG?C_??O?COG????I?J??Q??O?_@@??
↪@??????')
```

sage.graphs.generic_graph_pyx.**small_integer_to_graph6**(*n*)

Encodes a small integer (i.e. a number of vertices) as a graph6 string.

This helper function is named N [McK2015].

INPUT:

- `n` (integer)

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import small_integer_to_graph6
sage: small_integer_to_graph6(13)
'L'
sage: small_integer_to_graph6(136)
'~?AG'
```

sage.graphs.generic_graph_pyx.**spring_layout_fast**(*G*, *iterations=50*, *dim=2*, *vpos=None*, *rescale=True*, *height=False*, *by_component=False*, ***options*)

Spring force model layout

This function primarily acts as a wrapper around `run_spring()`, converting to and from raw C types.

This kind of speed cannot be achieved by naive Cythonification of the function alone, especially if we require a function call (let alone an object creation) every time we want to add a pair of doubles.

INPUT:

- `by_component` – a boolean

EXAMPLES:

```
sage: G = graphs.DodecahedralGraph()
sage: for i in range(10): G.add_cycle(list(range(100*i, 100*i+3)))
sage: from sage.graphs.generic_graph_pyx import spring_layout_fast
sage: pos = spring_layout_fast(G)
sage: pos[0]  # random
[0.00..., 0.03...]
sage: sorted(pos.keys()) == sorted(G)
True
```

With `split=True`, each component of G is layed out separately, placing them adjacent to each other. This is done because on a disconnected graph, the spring layout will push components further and further from each other without bound, resulting in very tight clumps for each component.

If the axis are scaled to fit the plot in a square, the horizontal distance may end up being "squished" due to the several adjacent components.

```
sage: G = graphs.DodecahedralGraph()
sage: for i in range(10): G.add_cycle(list(range(100*i, 100*i+3)))
sage: from sage.graphs.generic_graph_pyx import spring_layout_fast
sage: pos = spring_layout_fast(G, by_component = True)
sage: pos[0]  # random
[2.21..., -0.00...]
sage: len(pos) == G.order()
True
```

sage.graphs.generic_graph_pyx.**spring_layout_fast_split**(*G*, ***options*)

Graph each component of G separately, placing them adjacent to each other.

In ticket trac ticket #29522 the function was modified so that it can work with any layout method and renamed `layout_split`. Please use *`layout_split()`* from now on.

sage.graphs.generic_graph_pyx.**transitive_reduction_acyclic**(*G*)

Return the transitive reduction of an acyclic digraph.

INPUT:

- `G` – an acyclic digraph.

EXAMPLES:

```
sage: from sage.graphs.generic_graph_pyx import transitive_reduction_acyclic
sage: G = posets.BooleanLattice(4).hasse_diagram()
sage: G == transitive_reduction_acyclic(G.transitive_closure())
True
```

5.38 Orientations

This module implements several methods to compute orientations of undirected graphs subject to specific constraints (e.g., acyclic, strongly connected, etc.). It also implements some iterators over all these orientations.

This module contains the following methods

strong_orientations_iterator()	Return an iterator over all strong orientations of a graph G
random_orientation()	Return a random orientation of a graph G

5.38.1 Authors

- Kolja Knauer, Petru Valicov (2017-01-10) – initial version

5.38.2 Methods

sage.graphs.orientations.**random_orientation**(G)

Return a random orientation of a graph G.

An *orientation* of an undirected graph is a directed graph such that every edge is assigned a direction. Hence there are 2^m oriented digraphs for a simple graph with m edges.

INPUT:

- `G` – a Graph.

EXAMPLES:

```
sage: from sage.graphs.orientations import random_orientation
sage: G = graphs.PetersenGraph()
sage: D = random_orientation(G)
sage: D.order() == G.order(), D.size() == G.size()
(True, True)
```

See also:

- *orientations()*

sage.graphs.orientations.**strong_orientations_iterator**(G)

Returns an iterator over all strong orientations of a graph G.

A strong orientation of a graph is an orientation of its edges such that the obtained digraph is strongly connected (i.e. there exist a directed path between each pair of vertices).

ALGORITHM:

It is an adaptation of the algorithm published in [CGMRV16]. It runs in $O(mn)$ amortized time, where m is the number of edges and n is the number of vertices. The amortized time can be improved to $O(m)$ with a more involved method. In this function, first the graph is preprocessed and a spanning tree is generated. Then every orientation of the non-tree edges of the graph can be extended to at least one new strong orientation by orienting properly the edges of the spanning tree (this property is proved in [CGMRV16]). Therefore, this function generates all partial orientations of the non-tree edges and then launches a helper function corresponding to the generation algorithm described in [CGMRV16]. In order to avoid trivial symmetries, the orientation of an arbitrary edge is fixed before the start of the enumeration process.

INPUT:

- G – an undirected graph.

OUTPUT:

- an iterator which will produce all strong orientations of this graph.

Note: Works only for simple graphs (no multiple edges). To avoid symmetries an orientation of an arbitrary edge is fixed.

EXAMPLES:

A cycle has one possible (non-symmetric) strong orientation:

```
sage: g = graphs.CycleGraph(4)
sage: it = g.strong_orientations_iterator()
sage: len(list(it))
1
```

A tree cannot be strongly oriented:

```
sage: g = graphs.RandomTree(100)
sage: len(list(g.strong_orientations_iterator()))
0
```

Neither can be a disconnected graph:

```
sage: g = graphs.CompleteGraph(6)
sage: g.add_vertex(7)
sage: len(list(g.strong_orientations_iterator()))
0
```

5.39 Connectivity related functions

This module implements the connectivity based functions for graphs and digraphs. The methods in this module are also available as part of GenericGraph, DiGraph or Graph classes as aliases, and these methods can be accessed through this module or as class methods. Here is what the module can do:

For both directed and undirected graphs:

is_connected()	Check whether the (di)graph is connected.
connected_components()	Return the list of connected components
connected_components_number()	Return the number of connected components.
connected_components_subgraphs()	Return a list of connected components as graph objects.
connected_component_containing_vertex()	Return a list of the vertices connected to vertex.
connected_components_sizes()	Return the sizes of the connected components as a list.
blocks_and_cut_vertices()	Return the blocks and cut vertices of the graph.
blocks_and_cuts_tree()	Return the blocks-and-cuts tree of the graph.
is_cut_edge()	Return True if the input edge is a cut-edge or a bridge.
is_cut_vertex()	Check whether the input vertex is a cut-vertex.
edge_connectivity()	Return the edge connectivity of the graph.
vertex_connectivity()	Return the vertex connectivity of the graph.

For DiGraph:

is_strongly_connected()	Check whether the current DiGraph is strongly connected.
strongly_connected_components_digraph()	Return the digraph of the strongly connected components
strongly_connected_components_subgraphs()	Return the strongly connected components as a list of subgraphs.
strongly_connected_component_containing_vertex()	Return the strongly connected component containing a given vertex.
strong_articulation_points()	Return the strong articulation points of this digraph.

For undirected graphs:

bridges()	Returns a list of the bridges (or cut edges) of given undirected graph.
cleave()	Return the connected subgraphs separated by the input vertex cut.
is_triconnected()	Check whether the graph is triconnected.
spqr_tree()	Return a SPQR-tree representing the triconnected components of the graph.
spqr_tree_to_graph()	Return the graph represented by the SPQR-tree T.

5.39.1 Methods

class sage.graphs.connectivity.**TriconnectivitySPQR**

Bases: object

Decompose a graph into triconnected components and build SPQR-tree.

This class implements the algorithm proposed by Hopcroft and Tarjan in [Hopcroft1973], and later corrected by Gutwenger and Mutzel in [Gut2001], for finding the triconnected components of a biconnected graph. It then organizes these components into a SPQR-tree. See the:wikipedia:$SPQR_tree$.

A SPQR-tree is a tree data structure used to represent the triconnected components of a biconnected (multi)graph and the 2-vertex cuts separating them. A node of a SPQR-tree, and the graph associated with it, can be one of the following four types:

- "S" – the associated graph is a cycle with at least three vertices. "S" stands for series and is also called a polygon.
- "P" – the associated graph is a dipole graph, a multigraph with two vertices and three or more edges. "P" stands for parallel and the node is called a bond.
- "Q" – the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge.
- "R" – the associated graph is a 3-vertex-connected graph that is not a cycle or dipole. "R" stands for rigid.

The edges of the tree indicate the 2-vertex cuts of the graph.

INPUT:

- G – graph; if G is a *DiGraph*, the computation is done on the underlying *Graph* (i.e., ignoring edge orientation)
- check – boolean (default: True); indicates whether G needs to be tested for biconnectivity

See also:

- *sage.graphs.connectivity.spqr_tree()*
- *is_biconnected()*
- Wikipedia article SPQR_tree

EXAMPLES:

Example from the Wikipedia article SPQR_tree:

```
sage: from sage.graphs.connectivity import TriconnectivitySPQR
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G = Graph([(1, 2), (1, 4), (1, 8), (1, 12), (3, 4), (2, 3),
....: (2, 13), (3, 13), (4, 5), (4, 7), (5, 6), (5, 8), (5, 7), (6, 7),
....: (8, 11), (8, 9), (8, 12), (9, 10), (9, 11), (9, 12), (10, 12)])
sage: tric = TriconnectivitySPQR(G)
sage: T = tric.get_spqr_tree()
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
```

An example from [Hopcroft1973]:

```
sage: G = Graph([(1, 2), (1, 4), (1, 8), (1, 12), (1, 13), (2, 3),
....: (2, 13), (3, 4), (3, 13), (4, 5), (4, 7), (5, 6), (5, 7), (5, 8),
....: (6, 7), (8, 9), (8, 11), (8, 12), (9, 10), (9, 11), (9, 12),
....: (10, 11), (10, 12)])
sage: tric = TriconnectivitySPQR(G)
sage: T = tric.get_spqr_tree()
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: tric.print_triconnected_components()  # py2
Polygon: [(6, 7, None), (5, 6, None), (7, 5, 'newVEdge0')]
Bond: [(7, 5, 'newVEdge0'), (5, 7, 'newVEdge1'), (5, 7, None)]
Polygon: [(5, 7, 'newVEdge1'), (4, 7, None), (5, 4, 'newVEdge2')]
Bond: [(4, 5, None), (5, 4, 'newVEdge2'), (5, 4, 'newVEdge3')]
Polygon: [(5, 8, None), (5, 4, 'newVEdge3'), (1, 8, 'newVEdge8'), (1, 4,
↪'newVEdge9')]
Triconnected: [(8, 9, None), (9, 12, None), (9, 11, None), (8, 11, None), (10, 11,
↪ None), (9, 10, None), (10, 12, None), (8, 12, 'newVEdge5')]
Bond: [(8, 12, 'newVEdge5'), (12, 8, 'newVEdge6'), (8, 12, None)]
Polygon: [(1, 12, None), (12, 8, 'newVEdge6'), (1, 8, 'newVEdge7')]
Bond: [(1, 8, None), (1, 8, 'newVEdge7'), (1, 8, 'newVEdge8')]
Bond: [(1, 4, None), (1, 4, 'newVEdge9'), (1, 4, 'newVEdge10')]
Polygon: [(1, 4, 'newVEdge10'), (3, 4, None), (1, 3, 'newVEdge11')]
Triconnected: [(2, 3, None), (2, 13, None), (1, 2, None), (1, 3, 'newVEdge11'),␣
↪(1, 13, None), (3, 13, None)]
```

An example from [Gut2001]:

```
sage: G = Graph([(1, 2), (1, 4), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5),
....: (4, 6), (5, 7), (5, 8), (5, 14), (6, 8), (7, 14), (8, 9), (8, 10),
....: (8, 11), (8, 12), (9, 10), (10, 13), (10, 14), (10, 15), (10, 16),
....: (11, 12), (11, 13), (12, 13), (14, 15), (14, 16), (15, 16)])
sage: T = TriconnectivitySPQR(G).get_spqr_tree()
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
```

An example with multi-edges and accessing the triconnected components:

```
sage: G = Graph([(1, 2), (1, 5), (1, 5), (2, 3), (2, 3), (3, 4), (4, 5)],␣
↪multiedges=True)
sage: tric = TriconnectivitySPQR(G)
sage: T = tric.get_spqr_tree()
sage: G.is_isomorphic(spqr_tree_to_graph(T))
```

(continues on next page)

(continued from previous page)

```
True
sage: tric.print_triconnected_components()
Bond:  [(1, 5, None), (1, 5, None), (1, 5, 'newVEdge0')]
Bond:  [(2, 3, None), (2, 3, None), (2, 3, 'newVEdge1')]
Polygon:  [(4, 5, None), (1, 5, 'newVEdge0'), (3, 4, None), (2, 3, 'newVEdge1'),␣
↪(1, 2, None)]
```

An example of a triconnected graph:

```
sage: G = Graph([('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c',
↪ 'd')])
sage: T = TriconnectivitySPQR(G).get_spqr_tree()
sage: print(T.vertices())
[('R', Multi-graph on 4 vertices)]
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
```

An example of a directed graph with multi-edges:

```
sage: G = DiGraph([(1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (5, 1)])
sage: tric = TriconnectivitySPQR(G)
sage: tric.print_triconnected_components()
Bond:  [(1, 5, None), (5, 1, None), (1, 5, 'newVEdge0')]
Polygon:  [(4, 5, None), (1, 5, 'newVEdge0'), (3, 4, None), (2, 3, None), (1, 2,␣
↪None)]
```

Edge labels are preserved by the construction:

```
sage: G = Graph([(0, 1, '01'), (0, 4, '04'), (1, 2, '12'), (1, 5, '15'),
....: (2, 3, '23'), (2, 6, '26'), (3, 7, '37'), (4, 5, '45'),
....: (5, 6, '56'), (6, 7, 67)])
sage: T = TriconnectivitySPQR(G).get_spqr_tree()
sage: H = spqr_tree_to_graph(T)
sage: all(G.has_edge(e) for e in H.edge_iterator())
True
sage: all(H.has_edge(e) for e in G.edge_iterator())
True
```

get_spqr_tree()

Return an SPQR-tree representing the triconnected components of the graph.

An SPQR-tree is a tree data structure used to represent the triconnected components of a biconnected (multi)graph and the 2-vertex cuts separating them. A node of a SPQR-tree, and the graph associated with it, can be one of the following four types:

- `"S"` – the associated graph is a cycle with at least three vertices. `"S"` stands for `series`.
- `"P"` – the associated graph is a dipole graph, a multigraph with two vertices and three or more edges. `"P"` stands for `parallel`.
- `"Q"` – the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge.
- `"R"` – the associated graph is a 3-connected graph that is not a cycle or dipole. `"R"` stands for `rigid`.

The edges of the tree indicate the 2-vertex cuts of the graph.

OUTPUT:

`SPQR-tree` a tree whose vertices are labeled with the block's type and the subgraph of three-blocks in the decomposition.

EXAMPLES:

```
sage: from sage.graphs.connectivity import TriconnectivitySPQR
sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: K4 = graphs.CompleteGraph(4)
sage: all(u[1].is_isomorphic(K4) for u in Tree if u[0] == 'R')
True
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_path([0, G.add_vertex(), G.add_vertex(), 1])
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: C4 = graphs.CycleGraph(4)
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edge_iterator())
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = graphs.CycleGraph(6)
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: Tree.order()
1
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True
sage: G.add_edge(0, 3)
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: Tree.order()
3
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph([(0, 1)], multiedges=True)
sage: tric = TriconnectivitySPQR(G)
sage: Tree = tric.get_spqr_tree()
sage: Tree.vertices()
[('Q', Multi-graph on 2 vertices)]
```

(continues on next page)

(continued from previous page)

```
sage: G.add_edge(0, 1)
sage: Tree = TriconnectivitySPQR(G).get_spqr_tree()
sage: Tree.vertices()
[('P', Multi-graph on 2 vertices)]
```

get_triconnected_components()

Return the triconnected components as a list of tuples.

Each component is represented as a tuple of the type of the component and the list of edges of the component.

EXAMPLES:

```
sage: from sage.graphs.connectivity import TriconnectivitySPQR
sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_path([0, G.add_vertex(), G.add_vertex(), 1])
sage: tric = TriconnectivitySPQR(G)
sage: tric.get_triconnected_components()
[('Polygon', [(4, 5, None), (0, 4, None), (1, 5, None), (1, 0, 'newVEdge1')]),
('Polygon', [(6, 7, None), (0, 6, None), (1, 7, None), (1, 0, 'newVEdge3')]),
('Bond', [(1, 0, 'newVEdge1'), (1, 0, 'newVEdge3'), (1, 0, 'newVEdge4')]),
('Polygon', [(1, 3, None), (1, 0, 'newVEdge4'), (2, 3, None), (0, 2, None)])]
```

print_triconnected_components()

Print the type and list of edges of each component.

EXAMPLES:

An example from [Hopcroft1973]:

```
sage: from sage.graphs.connectivity import TriconnectivitySPQR
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G = Graph([(1, 2), (1, 4), (1, 8), (1, 12), (1, 13), (2, 3),
....: (2, 13), (3, 4), (3, 13), (4, 5), (4, 7), (5, 6), (5, 7), (5, 8),
....: (6, 7), (8, 9), (8, 11), (8, 12), (9, 10), (9, 11), (9, 12),
....: (10, 11), (10, 12)])
sage: tric = TriconnectivitySPQR(G)
sage: T = tric.get_spqr_tree()
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: tric.print_triconnected_components()  # py2
Polygon: [(6, 7, None), (5, 6, None), (7, 5, 'newVEdge0')]
Bond: [(7, 5, 'newVEdge0'), (5, 7, 'newVEdge1'), (5, 7, None)]
Polygon: [(5, 7, 'newVEdge1'), (4, 7, None), (5, 4, 'newVEdge2')]
Bond: [(4, 5, None), (5, 4, 'newVEdge2'), (5, 4, 'newVEdge3')]
Polygon: [(5, 8, None), (5, 4, 'newVEdge3'), (1, 8, 'newVEdge8'), (1, 4,
↪'newVEdge9')]
Triconnected: [(8, 9, None), (9, 12, None), (9, 11, None), (8, 11, None), (10,
↪ 11, None), (9, 10, None), (10, 12, None), (8, 12, 'newVEdge5')]
Bond: [(8, 12, 'newVEdge5'), (12, 8, 'newVEdge6'), (8, 12, None)]
Polygon: [(1, 12, None), (12, 8, 'newVEdge6'), (1, 8, 'newVEdge7')]
Bond: [(1, 8, None), (1, 8, 'newVEdge7'), (1, 8, 'newVEdge8')]
Bond: [(1, 4, None), (1, 4, 'newVEdge9'), (1, 4, 'newVEdge10')]
Polygon: [(1, 4, 'newVEdge10'), (3, 4, None), (1, 3, 'newVEdge11')]
Triconnected: [(2, 3, None), (2, 13, None), (1, 2, None), (1, 3, 'newVEdge11
↪'), (1, 13, None), (3, 13, None)]
sage: tric.print_triconnected_components()  # py3
```

(continues on next page)

(continued from previous page)

```
Triconnected: [(8, 9, None), (9, 12, None), (9, 11, None), (8, 11, None), (10,
↪ 11, None), (9, 10, None), (10, 12, None), (8, 12, 'newVEdge0')]
Bond: [(8, 12, None), (8, 12, 'newVEdge0'), (8, 12, 'newVEdge1')]
Polygon: [(6, 7, None), (5, 6, None), (7, 5, 'newVEdge2')]
Bond: [(7, 5, 'newVEdge2'), (5, 7, 'newVEdge3'), (5, 7, None)]
Polygon: [(5, 7, 'newVEdge3'), (4, 7, None), (5, 4, 'newVEdge4')]
Bond: [(5, 4, 'newVEdge4'), (4, 5, 'newVEdge5'), (4, 5, None)]
Polygon: [(4, 5, 'newVEdge5'), (5, 8, None), (1, 4, 'newVEdge9'), (1, 8,
↪'newVEdge10')]
Triconnected: [(1, 2, None), (2, 13, None), (1, 13, None), (3, 13, None), (2,␣
↪3, None), (1, 3, 'newVEdge7')]
Polygon: [(1, 3, 'newVEdge7'), (3, 4, None), (1, 4, 'newVEdge8')]
Bond: [(1, 4, None), (1, 4, 'newVEdge8'), (1, 4, 'newVEdge9')]
Bond: [(1, 8, None), (1, 8, 'newVEdge10'), (1, 8, 'newVEdge11')]
Polygon: [(8, 12, 'newVEdge1'), (1, 8, 'newVEdge11'), (1, 12, None)]
```

sage.graphs.connectivity.**blocks_and_cut_vertices**(*G*, *algorithm='Tarjan_Boost'*, *sort=False*)

Return the blocks and cut vertices of the graph.

In the case of a digraph, this computation is done on the underlying graph.

A cut vertex is one whose deletion increases the number of connected components. A block is a maximal induced subgraph which itself has no cut vertices. Two distinct blocks cannot overlap in more than a single cut vertex.

INPUT:

- `algorithm` – string (default: `"Tarjan_Boost"`); the algorithm to use among:
  - `"Tarjan_Boost"` (default) – Tarjan's algorithm (Boost implementation)
  - `"Tarjan_Sage"` – Tarjan's algorithm (Sage implementation)
- `sort` – boolean (default: `False`); whether to sort vertices inside the components and the list of cut vertices **currently only available for ‘‘”Tarjan_Sage”‘‘**

OUTPUT: `(B, C)`, where `B` is a list of blocks - each is a list of vertices and the blocks are the corresponding induced subgraphs - and `C` is a list of cut vertices.

ALGORITHM:

We implement the algorithm proposed by Tarjan in [Tarjan72]. The original version is recursive. We emulate the recursion using a stack.

See also:

- *blocks_and_cuts_tree()*
- *sage.graphs.base.boost_graph.blocks_and_cut_vertices()*
- *is_biconnected()*
- *bridges()*

EXAMPLES:

We construct a trivial example of a graph with one cut vertex:

```
sage: from sage.graphs.connectivity import blocks_and_cut_vertices
sage: rings = graphs.CycleGraph(10)
sage: rings.merge_vertices([0, 5])
sage: blocks_and_cut_vertices(rings)
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
sage: rings.blocks_and_cut_vertices()
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
sage: B, C = blocks_and_cut_vertices(rings, algorithm="Tarjan_Sage", sort=True)
sage: B, C
([[0, 1, 2, 3, 4], [0, 6, 7, 8, 9]], [0])
sage: B2, C2 = blocks_and_cut_vertices(rings, algorithm="Tarjan_Sage", sort=False)
sage: Set(map(Set, B)) == Set(map(Set, B2)) and set(C) == set(C2)
True
```

The Petersen graph is biconnected, hence has no cut vertices:

```
sage: blocks_and_cut_vertices(graphs.PetersenGraph())
([[0, 1, 4, 5, 2, 6, 3, 7, 8, 9]], [])
```

Decomposing paths to pairs:

```
sage: g = graphs.PathGraph(4) + graphs.PathGraph(5)
sage: blocks_and_cut_vertices(g)
([[2, 3], [1, 2], [0, 1], [7, 8], [6, 7], [5, 6], [4, 5]], [1, 2, 5, 6, 7])
```

A disconnected graph:

```
sage: g = Graph({1: {2: 28, 3: 10}, 2: {1: 10, 3: 16}, 4: {}, 5: {6: 3, 7: 10, 8:␣
↪4}})
sage: blocks_and_cut_vertices(g)
([[1, 2, 3], [5, 6], [5, 7], [5, 8], [4]], [5])
```

A directed graph with Boost's algorithm (trac ticket #25994):

```
sage: rings = graphs.CycleGraph(10)
sage: rings.merge_vertices([0, 5])
sage: rings = rings.to_directed()
sage: blocks_and_cut_vertices(rings, algorithm="Tarjan_Boost")
([[0, 1, 4, 2, 3], [0, 6, 9, 7, 8]], [0])
```

sage.graphs.connectivity.**blocks_and_cuts_tree**(*G*)

Return the blocks-and-cuts tree of `self`.

This new graph has two different kinds of vertices, some representing the blocks (type B) and some other the cut vertices of the graph (type C).

There is an edge between a vertex u of type B and a vertex v of type C if the cut-vertex corresponding to v is in the block corresponding to u.

The resulting graph is a tree, with the additional characteristic property that the distance between two leaves is even. When `self` is not connected, the resulting graph is a forest.

When `self` is biconnected, the tree is reduced to a single node of type B.

We referred to [HarPri] and [Gallai] for blocks and cuts tree.

See also:

- *blocks_and_cut_vertices()*

- *is_biconnected()*

EXAMPLES:

```
sage: from sage.graphs.connectivity import blocks_and_cuts_tree
sage: T = blocks_and_cuts_tree(graphs.KrackhardtKiteGraph()); T
Graph on 5 vertices
sage: T.is_isomorphic(graphs.PathGraph(5))
True
sage: from sage.graphs.connectivity import blocks_and_cuts_tree
sage: T = graphs.KrackhardtKiteGraph().blocks_and_cuts_tree(); T
Graph on 5 vertices
```

The distance between two leaves is even:

```
sage: T = blocks_and_cuts_tree(graphs.RandomTree(40))
sage: T.is_tree()
True
sage: leaves = [v for v in T if T.degree(v) == 1]
sage: all(T.distance(u,v) % 2 == 0 for u in leaves for v in leaves)
True
```

The tree of a biconnected graph has a single vertex, of type B:

```
sage: T = blocks_and_cuts_tree(graphs.PetersenGraph())
sage: T.vertices()
[('B', (0, 1, 4, 5, 2, 6, 3, 7, 8, 9))]
```

sage.graphs.connectivity.**bridges**(*G*, *labels=True*)

Return a list of the bridges (or cut edges).

A bridge is an edge whose deletion disconnects the undirected graph. A disconnected graph has no bridge.

INPUT:

- `labels` – boolean (default: `True`); if `False`, each bridge is a tuple (u, v) of vertices

EXAMPLES:

```
sage: from sage.graphs.connectivity import bridges
sage: from sage.graphs.connectivity import is_connected
sage: g = 2 * graphs.PetersenGraph()
sage: g.add_edge(1, 10)
sage: is_connected(g)
True
sage: bridges(g)
[(1, 10, None)]
sage: g.bridges()
[(1, 10, None)]
```

sage.graphs.connectivity.**cleave**(*G*, *cut_vertices=None*, *virtual_edges=True*, *solver=None*, *verbose=0*)

Return the connected subgraphs separated by the input vertex cut.

Given a connected (multi)graph G and a vertex cut X, this method computes the list of subgraphs of G induced by each connected component c of $G \setminus X$ plus X, i.e., $G[c \cup X]$.

INPUT:

- `G` – a Graph.

- `cut_vertices` – iterable container of vertices (default: `None`); a set of vertices representing a vertex cut of `G`. If no vertex cut is given, the method will compute one via a call to *`vertex_connectivity()`*.
- `virtual_edges` – boolean (default: `True`); whether to add virtual edges to the sides of the cut or not. A virtual edge is an edge between a pair of vertices of the cut that are not connected by an edge in `G`.
- `solver` – string (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `sage.numerical.mip.MixedIntegerLinearProgram.solve()` of the class `sage.numerical.mip.MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT: A triple (S, C, f), where

- S is a list of the graphs that are sides of the vertex cut.
- C is the graph of the cocycles. For each pair of vertices of the cut, if there exists an edge between them, C has one copy of each edge connecting them in `G` per sides of the cut plus one extra copy. Furthermore, when `virtual_edges == True`, if a pair of vertices of the cut is not connected by an edge in `G`, then it has one virtual edge between them per sides of the cut.
- f is the complement of the subgraph of `G` induced by the vertex cut. Hence, its vertex set is the vertex cut, and its edge set is the set of virtual edges (i.e., edges between pairs of vertices of the cut that are not connected by an edge in `G`). When `virtual_edges == False`, the edge set is empty.

EXAMPLES:

If there is an edge between cut vertices:

```
sage: from sage.graphs.connectivity import cleave
sage: G = Graph(2)
sage: for _ in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: S1,C1,f1 = cleave(G, cut_vertices=[0, 1])
sage: [g.order() for g in S1]
[4, 4, 4]
sage: C1.order(), C1.size()
(2, 4)
sage: f1.vertices(), f1.edges()
([0, 1], [])
```

If `virtual_edges == False` and there is an edge between cut vertices:

```
sage: G.subgraph([0, 1]).complement() == Graph([[0, 1], []])
True
sage: S2,C2,f2 = cleave(G, cut_vertices=[0, 1], virtual_edges=False)
sage: (S1 == S2, C1 == C2, f1 == f2)
(True, True, True)
```

If cut vertices doesn't have edge between them:

```
sage: G.delete_edge(0, 1)
sage: S1,C1,f1 = cleave(G, cut_vertices=[0, 1])
sage: [g.order() for g in S1]
[4, 4, 4]
sage: C1.order(), C1.size()
(2, 3)
sage: f1.vertices(), f1.edges()
([0, 1], [(0, 1, None)])
```

If `virtual_edges == False` and the cut vertices are not connected by an edge:

```
sage: G.subgraph([0, 1]).complement() == Graph([[0, 1], []])
False
sage: S2,C2,f2 = cleave(G, cut_vertices=[0, 1], virtual_edges=False)
sage: [g.order() for g in S2]
[4, 4, 4]
sage: C2.order(), C2.size()
(2, 0)
sage: f2.vertices(), f2.edges()
([0, 1], [])
sage: (S1 == S2, C1 == C2, f1 == f2)
(False, False, False)
```

If G is a biconnected multigraph:

```
sage: G = graphs.CompleteBipartiteGraph(2, 3)
sage: G.add_edge(2, 3)
sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edge_iterator())
sage: G.add_edges([(0, 1), (0, 1), (0, 1)])
sage: S,C,f = cleave(G, cut_vertices=[0, 1])
sage: for g in S:
....:     print(g.edges(labels=0))
[(0, 1), (0, 1), (0, 1), (0, 2), (0, 2), (0, 3), (0, 3), (1, 2), (1, 2), (1, 3),␣
↪(1, 3), (2, 3), (2, 3)]
[(0, 1), (0, 1), (0, 1), (0, 4), (0, 4), (1, 4), (1, 4)]
```

sage.graphs.connectivity.**connected_component_containing_vertex**(*G*, *vertex*, *sort=True*)

Return a list of the vertices connected to vertex.

INPUT:

- `G` – the input graph
- `v` – the vertex to search for
- `sort` – boolean (default `True`); whether to sort vertices inside the component

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_component_containing_vertex
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_component_containing_vertex(G, 0)
[0, 1, 2, 3]
sage: G.connected_component_containing_vertex(0)
[0, 1, 2, 3]
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_component_containing_vertex(D, 0)
[0, 1, 2, 3]
```

sage.graphs.connectivity.**connected_components**(*G*, *sort=True*)

Return the list of connected components.

This returns a list of lists of vertices, each list representing a connected component. The list is ordered from largest to smallest component.

INPUT:

- `G` – the input graph

- sort – boolean (default True); whether to sort vertices inside each component

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components(G)
[[0, 1, 2, 3], [4, 5, 6]]
sage: G.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components(D)
[[0, 1, 2, 3], [4, 5, 6]]
```

sage.graphs.connectivity.**connected_components_number**(*G*)

Return the number of connected components.

INPUT:

- G – the input graph

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_number
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components_number(G)
2
sage: G.connected_components_number()
2
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: connected_components_number(D)
2
```

sage.graphs.connectivity.**connected_components_sizes**(*G*)

Return the sizes of the connected components as a list.

The list is sorted from largest to lower values.

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_sizes
sage: for x in graphs(3):
....:     print(connected_components_sizes(x))
[1, 1, 1]
[2, 1]
[3]
[3]
sage: for x in graphs(3):
....:     print(x.connected_components_sizes())
[1, 1, 1]
[2, 1]
[3]
[3]
```

sage.graphs.connectivity.**connected_components_subgraphs**(*G*)

Return a list of connected components as graph objects.

EXAMPLES:

```
sage: from sage.graphs.connectivity import connected_components_subgraphs
sage: G = Graph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: L = connected_components_subgraphs(G)
sage: graphs_list.show_graphs(L)
sage: D = DiGraph({0: [1, 3], 1: [2], 2: [3], 4: [5, 6], 5: [6]})
sage: L = connected_components_subgraphs(D)
sage: graphs_list.show_graphs(L)
sage: L = D.connected_components_subgraphs()
sage: graphs_list.show_graphs(L)
```

sage.graphs.connectivity.**edge_connectivity**(*G*, *value_only=True*, *implementation=None*, *use_edge_labels=False*, *vertices=False*, *solver=None*, *verbose=0*)

Return the edge connectivity of the graph.

For more information, see the Wikipedia article Connectivity_(graph_theory).

Note: When the graph is a directed graph, this method actually computes the *strong* connectivity, (i.e. a directed graph is strongly k-connected if there are k disjoint paths between any two vertices u, v). If you do not want to consider strong connectivity, the best is probably to convert your `DiGraph` object to a `Graph` object, and compute the connectivity of this other graph.

INPUT:

- `G` – the input Sage (Di)Graph
- `value_only` – boolean (default: `True`)
  - When set to `True` (default), only the value is returned.
  - When set to `False`, both the value and a minimum vertex cut are returned.
- `implementation` – string (default: `None`); selects an implementation:
  - `None` (default) – selects the best implementation available
  - `"boost"` – use the Boost graph library (which is much more efficient). It is not available when `edge_labels=True`, and it is unreliable for directed graphs (see trac ticket #18753).
  - **`"Sage"` – use Sage's implementation based on integer linear** programming
- `use_edge_labels` – boolean (default: `False`)
  - When set to `True`, computes a weighted minimum cut where each edge has a weight defined by its label. (If an edge has no label, 1 is assumed.). Implies `boost` = `False`.
  - When set to `False`, each edge has weight 1.
- `vertices` – boolean (default: `False`)
  - When set to `True`, also returns the two sets of vertices that are disconnected by the cut. Implies `value_only=False`.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used (ignored if `implementation='boost'`). If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic application on the PappusGraph:

```
sage: from sage.graphs.connectivity import edge_connectivity
sage: g = graphs.PappusGraph()
sage: edge_connectivity(g)
3
sage: g.edge_connectivity()
3
```

The edge connectivity of a complete graph is its minimum degree, and one of the two parts of the bipartition is reduced to only one vertex. The graph of the cut edges is isomorphic to a Star graph:

```
sage: g = graphs.CompleteGraph(5)
sage: [ value, edges, [ setA, setB ]] = edge_connectivity(g,vertices=True)
sage: value
4
sage: len(setA) == 1 or len(setB) == 1
True
sage: cut = Graph()
sage: cut.add_edges(edges)
sage: cut.is_isomorphic(graphs.StarGraph(4))
True
```

Even if obviously in any graph we know that the edge connectivity is less than the minimum degree of the graph:

```
sage: g = graphs.RandomGNP(10,.3)
sage: min(g.degree()) >= edge_connectivity(g)
True
```

If we build a tree then assign to its edges a random value, the minimum cut will be the edge with minimum value:

```
sage: tree = graphs.RandomTree(10)
sage: for u,v in tree.edge_iterator(labels=None):
....:     tree.set_edge_label(u, v, random())
sage: minimum = min(tree.edge_labels())
sage: [_, [(_, _, l)]] = edge_connectivity(tree, value_only=False, use_edge_
↪labels=True)
sage: l == minimum
True
```

When `value_only=True` and `implementation="sage"`, this function is optimized for small connectivity values and does not need to build a linear program.

It is the case for graphs which are not connected

```
sage: g = 2 * graphs.PetersenGraph()
sage: edge_connectivity(g, implementation="sage")
0.0
```

For directed graphs, the strong connectivity is tested through the dedicated function:

```
sage: g = digraphs.ButterflyGraph(3)
sage: edge_connectivity(g, implementation="sage")
0.0
```

We check that the result with Boost is the same as the result without Boost:

```
sage: g = graphs.RandomGNP(15, .3)
sage: edge_connectivity(g, implementation="boost") == edge_connectivity(g,␣
↪implementation="sage")
True
```

Boost interface also works with directed graphs:

```
sage: edge_connectivity(digraphs.Circuit(10), implementation="boost",␣
↪vertices=True)
[1, [(0, 1)], [{0}, {1, 2, 3, 4, 5, 6, 7, 8, 9}]]
```

However, the Boost algorithm is not reliable if the input is directed (see trac ticket #18753):

```
sage: g = digraphs.Path(3)
sage: edge_connectivity(g)
0.0
sage: edge_connectivity(g, implementation="boost")
1
sage: g.add_edge(1, 0)
sage: edge_connectivity(g)
0.0
sage: edge_connectivity(g, implementation="boost")
0
```

sage.graphs.connectivity.**is_connected**(*G*)

Check whether the (di)graph is connected.

Note that in a graph, path connected is equivalent to connected.

INPUT:

- `G` – the input graph

See also:

- *is_biconnected()*

EXAMPLES:

```
sage: from sage.graphs.connectivity import is_connected
sage: G = Graph({0: [1, 2], 1: [2], 3: [4, 5], 4: [5]})
sage: is_connected(G)
False
sage: G.is_connected()
False
sage: G.add_edge(0,3)
sage: is_connected(G)
True
sage: D = DiGraph({0: [1, 2], 1: [2], 3: [4, 5], 4: [5]})
sage: is_connected(D)
False
sage: D.add_edge(0, 3)
sage: is_connected(D)
True
sage: D = DiGraph({1: [0], 2: [0]})
sage: is_connected(D)
True
```

sage.graphs.connectivity.**is_cut_edge**(*G*, *u*, *v=None*, *label=None*)

Returns True if the input edge is a cut-edge or a bridge.

A cut edge (or bridge) is an edge that when removed increases the number of connected components. This function works with simple graphs as well as graphs with loops and multiedges. In a digraph, a cut edge is an edge that when removed increases the number of (weakly) connected components.

INPUT: The following forms are accepted

- is_cut_edge(G, 1, 2)
- is_cut_edge(G, (1, 2))
- is_cut_edge(G, 1, 2, 'label')
- is_cut_edge(G, (1, 2, 'label'))

OUTPUT:

- Returns True if (u,v) is a cut edge, False otherwise

EXAMPLES:

```
sage: from sage.graphs.connectivity import is_cut_edge
sage: G = graphs.CompleteGraph(4)
sage: is_cut_edge(G,0,2)
False
sage: G.is_cut_edge(0,2)
False

sage: G = graphs.CompleteGraph(4)
sage: G.add_edge((0,5,'silly'))
sage: is_cut_edge(G,(0,5,'silly'))
True

sage: G = Graph([[0,1],[0,2],[3,4],[4,5],[3,5]])
sage: is_cut_edge(G,(0,1))
True

sage: G = Graph([[0,1],[0,2],[1,1]], loops = True)
sage: is_cut_edge(G,(1,1))
False

sage: G = digraphs.Circuit(5)
sage: is_cut_edge(G,(0,1))
False

sage: G = graphs.CompleteGraph(6)
sage: is_cut_edge(G,(0,7))
Traceback (most recent call last):
...
ValueError: edge not in graph
```

sage.graphs.connectivity.**is_cut_vertex**(*G*, *u*, *weak=False*)

Check whether the input vertex is a cut-vertex.

A vertex is a cut-vertex if its removal from the (di)graph increases the number of (strongly) connected components. Isolated vertices or leafs are not cut-vertices. This function works with simple graphs as well as graphs with loops and multiple edges.

INPUT:

- G – a Sage (Di)Graph

- u – a vertex
- weak – boolean (default: False); whether the connectivity of directed graphs is to be taken in the weak sense, that is ignoring edges orientations

OUTPUT:

Return True if u is a cut-vertex, and False otherwise.

EXAMPLES:

Giving a LollipopGraph(4,2), that is a complete graph with 4 vertices with a pending edge:

```
sage: from sage.graphs.connectivity import is_cut_vertex
sage: G = graphs.LollipopGraph(4, 2)
sage: is_cut_vertex(G, 0)
False
sage: is_cut_vertex(G, 3)
True
sage: G.is_cut_vertex(3)
True
```

Comparing the weak and strong connectivity of a digraph:

```
sage: from sage.graphs.connectivity import is_strongly_connected
sage: D = digraphs.Circuit(6)
sage: is_strongly_connected(D)
True
sage: is_cut_vertex(D, 2)
True
sage: is_cut_vertex(D, 2, weak=True)
False
```

Giving a vertex that is not in the graph:

```
sage: G = graphs.CompleteGraph(4)
sage: is_cut_vertex(G, 7)
Traceback (most recent call last):
...
ValueError: vertex (7) is not a vertex of the graph
```

sage.graphs.connectivity.**is_strongly_connected**(G)

Check whether the current DiGraph is strongly connected.

EXAMPLES:

The circuit is obviously strongly connected:

```
sage: from sage.graphs.connectivity import is_strongly_connected
sage: g = digraphs.Circuit(5)
sage: is_strongly_connected(g)
True
sage: g.is_strongly_connected()
True
```

But a transitive triangle is not:

```
sage: g = DiGraph({0: [1, 2], 1: [2]})
sage: is_strongly_connected(g)
False
```

sage.graphs.connectivity.**is_triconnected**(*G*)

Check whether the graph is triconnected.

A triconnected graph is a connected graph on 3 or more vertices that is not broken into disconnected pieces by deleting any pair of vertices.

EXAMPLES:

The Petersen graph is triconnected:

```
sage: G = graphs.PetersenGraph()
sage: G.is_triconnected()
True
```

But a 2D grid is not:

```
sage: G = graphs.Grid2dGraph(3, 3)
sage: G.is_triconnected()
False
```

By convention, a cycle of order 3 is triconnected:

```
sage: G = graphs.CycleGraph(3)
sage: G.is_triconnected()
True
```

But cycles of order 4 and more are not:

```
sage: [graphs.CycleGraph(i).is_triconnected() for i in range(4, 8)]
[False, False, False, False]
```

Comparing different methods on random graphs that are not always triconnected:

```
sage: G = graphs.RandomBarabasiAlbert(50, 3)
sage: G.is_triconnected() == G.vertex_connectivity(k=3)
True
```

See also:

- *is_connected()*
- *is_biconnected()*
- *spqr_tree()*
- Wikipedia article SPQR_tree

sage.graphs.connectivity.**spqr_tree**(*G*, *algorithm='Hopcroft_Tarjan'*, *solver=None*, *verbose=0*)

Return an SPQR-tree representing the triconnected components of the graph.

An SPQR-tree is a tree data structure used to represent the triconnected components of a biconnected (multi)graph and the 2-vertex cuts separating them. A node of a SPQR-tree, and the graph associated with it, can be one of the following four types:

- `"S"` – the associated graph is a cycle with at least three vertices. `"S"` stands for `series`.
- `"P"` – the associated graph is a dipole graph, a multigraph with two vertices and three or more edges. `"P"` stands for `parallel`.

- "Q" – the associated graph has a single real edge. This trivial case is necessary to handle the graph that has only one edge.
- "R" – the associated graph is a 3-connected graph that is not a cycle or dipole. "R" stands for rigid.

This method decomposes a biconnected graph into cycles, cocycles, and 3-connected blocks summed over cocycles, and arranges them as a SPQR-tree. More precisely, it splits the graph at each of its 2-vertex cuts, giving a unique decomposition into 3-connected blocks, cycles and cocycles. The cocycles are dipole graphs with one edge per real edge between the included vertices and one additional (virtual) edge per connected component resulting from deletion of the vertices in the cut. See the Wikipedia article SPQR_tree.

INPUT:

- G – the input graph
- algorithm – string (default: "Hopcroft_Tarjan"); the algorithm to use among:
  - "Hopcroft_Tarjan" (default) – use the algorithm proposed by Hopcroft and Tarjan in [Hopcroft1973] and later corrected by Gutwenger and Mutzel in [Gut2001]. See *TriconnectivitySPQR*.
  - "cleave" – using method *cleave()*
- solver – string (default: None); specifies a Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method sage.numerical.mip.MixedIntegerLinearProgram.solve() of the class sage.numerical.mip.MixedIntegerLinearProgram.
- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.

OUTPUT: SPQR-tree a tree whose vertices are labeled with the block's type and the subgraph of three-blocks in the decomposition.

EXAMPLES:

```
sage: from sage.graphs.connectivity import spqr_tree
sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: Tree = spqr_tree(G)
sage: Tree.order()
4
sage: K4 = graphs.CompleteGraph(4)
sage: all(u[1].is_isomorphic(K4) for u in Tree if u[0] == 'R')
True
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph(2)
sage: for i in range(3):
....:     G.add_path([0, G.add_vertex(), G.add_vertex(), 1])
sage: Tree = spqr_tree(G)
sage: Tree.order()
4
sage: C4 = graphs.CycleGraph(4)
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True
```

(continues on next page)

(continued from previous page)

```
sage: G.allow_multiple_edges(True)
sage: G.add_edges(G.edge_iterator())
sage: Tree = spqr_tree(G)
sage: Tree.order()
13
sage: all(u[1].is_isomorphic(C4) for u in Tree if u[0] == 'S')
True
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = graphs.CycleGraph(6)
sage: Tree = spqr_tree(G)
sage: Tree.order()
1
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True
sage: G.add_edge(0, 3)
sage: Tree = spqr_tree(G)
sage: Tree.order()
3
sage: G.is_isomorphic(spqr_tree_to_graph(Tree))
True

sage: G = Graph('LlCG{O@?GBoMw?')
sage: T = spqr_tree(G, algorithm="Hopcroft_Tarjan")
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: T2 = spqr_tree(G, algorithm='cleave')
sage: G.is_isomorphic(spqr_tree_to_graph(T2))
True

sage: G = Graph([(0, 1)], multiedges=True)
sage: T = spqr_tree(G, algorithm='cleave')
sage: T.vertices()
[('Q', Multi-graph on 2 vertices)]
sage: G.is_isomorphic(spqr_tree_to_graph(T))
True
sage: T = spqr_tree(G, algorithm='Hopcroft_Tarjan')
sage: T.vertices()
[('Q', Multi-graph on 2 vertices)]
sage: G.add_edge(0, 1)
sage: spqr_tree(G, algorithm='cleave').vertices()
[('P', Multi-graph on 2 vertices)]

sage: from collections import Counter
sage: G = graphs.PetersenGraph()
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: Counter(u[0] for u in T)
Counter({'R': 1})
sage: T = G.spqr_tree(algorithm="cleave")
sage: Counter(u[0] for u in T)
Counter({'R': 1})
sage: for u,v in list(G.edges(labels=False, sort=False)):
....:     G.add_path([u, G.add_vertex(), G.add_vertex(), v])
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 15), ('R', 1), ('S', 15)]
```

(continues on next page)

(continued from previous page)

```
sage: T = G.spqr_tree(algorithm="cleave")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 15), ('R', 1), ('S', 15)]
sage: for u,v in list(G.edges(labels=False, sort=False)):
....:     G.add_path([u, G.add_vertex(), G.add_vertex(), v])
sage: T = G.spqr_tree(algorithm="Hopcroft_Tarjan")
sage: sorted(Counter(u[0] for u in T).items())
[('P', 60), ('R', 1), ('S', 75)]
sage: T = G.spqr_tree(algorithm="cleave")       # long time
sage: sorted(Counter(u[0] for u in T).items())  # long time
[('P', 60), ('R', 1), ('S', 75)]
```

sage.graphs.connectivity.**spqr_tree_to_graph**(*T*)

Return the graph represented by the SPQR-tree T.

The main purpose of this method is to test *spqr_tree()*.

INPUT:

- T – a SPQR tree as returned by *spqr_tree()*.

OUTPUT: a (multi) graph

EXAMPLES:

Wikipedia article SPQR_tree reference paper example:

```
sage: from sage.graphs.connectivity import spqr_tree
sage: from sage.graphs.connectivity import spqr_tree_to_graph
sage: G = Graph([(1, 2), (1, 4), (1, 8), (1, 12), (3, 4), (2, 3),
....: (2, 13), (3, 13), (4, 5), (4, 7), (5, 6), (5, 8), (5, 7), (6, 7),
....: (8, 11), (8, 9), (8, 12), (9, 10), (9, 11), (9, 12), (10, 12)])
sage: T = spqr_tree(G)
sage: H = spqr_tree_to_graph(T)
sage: H.is_isomorphic(G)
True
```

A small multigraph

```
sage: G = Graph([(0, 2), (0, 2), (1, 3), (2, 3)], multiedges=True)
sage: for i in range(3):
....:     G.add_clique([0, 1, G.add_vertex(), G.add_vertex()])
sage: for i in range(3):
....:     G.add_clique([2, 3, G.add_vertex(), G.add_vertex()])
sage: T = spqr_tree(G)
sage: H = spqr_tree_to_graph(T)
sage: H.is_isomorphic(G)
True
```

sage.graphs.connectivity.**strong_articulation_points**(*G*)

Return the strong articulation points of this digraph.

A vertex is a strong articulation point if its deletion increases the number of strongly connected components. This method implements the algorithm described in [ILS2012]. The time complexity is dominated by the time complexity of the immediate dominators finding algorithm.

OUTPUT: The list of strong articulation points.

EXAMPLES:

Two cliques sharing a vertex:

```
sage: from sage.graphs.connectivity import strong_articulation_points
sage: D = digraphs.Complete(4)
sage: D.add_clique([3, 4, 5, 6])
sage: strong_articulation_points(D)
[3]
sage: D.strong_articulation_points()
[3]
```

Two cliques connected by some arcs:

```
sage: D = digraphs.Complete(4) * 2
sage: D.add_edges([(0, 4), (7, 3)])
sage: sorted(strong_articulation_points(D))
[0, 3, 4, 7]
sage: D.add_edge(1, 5)
sage: sorted(strong_articulation_points(D))
[3, 7]
sage: D.add_edge(6, 2)
sage: strong_articulation_points(D)
[]
```

See also:

- *strongly_connected_components()*
- *dominator_tree()*

sage.graphs.connectivity.**strongly_connected_component_containing_vertex**(*G*, *v*)

Return the strongly connected component containing a given vertex

INPUT:

- G – the input DiGraph
- v – a vertex

EXAMPLES:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: from sage.graphs.connectivity import strongly_connected_component_
↪containing_vertex
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: strongly_connected_component_containing_vertex(d, 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: d.strongly_connected_component_containing_vertex(0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
sage: g = DiGraph([(0, 1), (1, 0), (1, 2), (2, 3), (3, 2)])
sage: strongly_connected_component_containing_vertex(g, 0)
[0, 1]
```

sage.graphs.connectivity.**strongly_connected_components_digraph**(*G*, *keep_labels=False*)

Return the digraph of the strongly connected components

The digraph of the strongly connected components of a graph G has a vertex per strongly connected component included in G. There is an edge from a component C_1 to a component C_2 if there is an edge in G from a vertex $u_1 \in C_1$ to a vertex $u_2 \in C_2$.

INPUT:

- `G` – the input DiGraph
- `keep_labels` – boolean (default: `False`); when `keep_labels=True`, the resulting digraph has an edge from a component C_i to a component C_j for each edge in G from a vertex $u_i \in C_i$ to a vertex $u_j \in C_j$. Hence the resulting digraph may have loops and multiple edges. However, edges in the result with same source, target, and label are not duplicated (see examples below). When `keep_labels=False`, the return digraph is simple, so without loops nor multiple edges, and edges are unlabelled.

EXAMPLES:

Such a digraph is always acyclic:

```
sage: from sage.graphs.connectivity import strongly_connected_components_digraph
sage: g = digraphs.RandomDirectedGNP(15, .1)
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: scc_digraph.is_directed_acyclic()
True
sage: scc_digraph = g.strongly_connected_components_digraph()
sage: scc_digraph.is_directed_acyclic()
True
```

The vertices of the digraph of strongly connected components are exactly the strongly connected components:

```
sage: g = digraphs.ButterflyGraph(2)
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: g.is_directed_acyclic()
True
sage: V_scc = list(scc_digraph)
sage: all(Set(scc) in V_scc for scc in g.strongly_connected_components())
True
```

The following digraph has three strongly connected components, and the digraph of those is a `TransitiveTournament()`:

```
sage: g = DiGraph({0: {1: "01", 2: "02", 3: "03"}, 1: {2: "12"}, 2:{1: "21", 3:
↪"23"}})
sage: scc_digraph = strongly_connected_components_digraph(g)
sage: scc_digraph.is_isomorphic(digraphs.TransitiveTournament(3))
True
```

By default, the labels are discarded, and the result has no loops nor multiple edges. If `keep_labels` is `True`, then the labels are kept, and the result is a multi digraph, possibly with multiple edges and loops. However, edges in the result with same source, target, and label are not duplicated (see the edges from 0 to the strongly connected component $\{1, 2\}$ below):

```
sage: g = DiGraph({0: {1: "0-12", 2: "0-12", 3: "0-3"}, 1: {2: "1-2", 3: "1-3"},␣
↪2: {1: "2-1", 3: "2-3"}})
sage: g.order(), g.size()
(4, 7)
sage: scc_digraph = strongly_connected_components_digraph(g, keep_labels=True)
sage: (scc_digraph.order(), scc_digraph.size())
(3, 6)
```

(continues on next page)

(continued from previous page)

```
sage: set(g.edge_labels()) == set(scc_digraph.edge_labels())
True
```

sage.graphs.connectivity.**strongly_connected_components_subgraphs**(*G*)

Return the strongly connected components as a list of subgraphs.

EXAMPLES:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: from sage.graphs.connectivity import strongly_connected_components_subgraphs
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: strongly_connected_components_subgraphs(d)
[Subgraph of (Petersen graph): Digraph on 10 vertices]
sage: d.strongly_connected_components_subgraphs()
[Subgraph of (Petersen graph): Digraph on 10 vertices]
```

```
sage: g = DiGraph([(0, 1), (1, 0), (1, 2), (2, 3), (3, 2)])
sage: strongly_connected_components_subgraphs(g)
[Subgraph of (): Digraph on 2 vertices, Subgraph of (): Digraph on 2 vertices]
```

sage.graphs.connectivity.**vertex_connectivity**(*G*, *value_only=True*, *sets=False*, *k=None*, *solver=None*, *verbose=0*)

Return the vertex connectivity of the graph.

For more information, see the Wikipedia article Connectivity_(graph_theory) and the Wikipedia article K-vertex-connected_graph.

Note:

- When the graph is directed, this method actually computes the *strong* connectivity, (i.e. a directed graph is strongly k-connected if there are k vertex disjoint paths between any two vertices u, v). If you do not want to consider strong connectivity, the best is probably to convert your `DiGraph` object to a `Graph` object, and compute the connectivity of this other graph.
- By convention, a complete graph on n vertices is $n-1$ connected. In this case, no certificate can be given as there is no pair of vertices split by a cut of order $k-1$. For this reason, the certificates returned in this situation are empty.

INPUT:

- `G` – the input Sage (Di)Graph
- `value_only` – boolean (default: `True`)
  - When set to `True` (default), only the value is returned.
  - When set to `False`, both the value and a minimum vertex cut are returned.
- **`sets` – boolean (default: `False`); whether to also return the two** sets of vertices that are disconnected by the cut (implies `value_only=False`)
- `k` – integer (default: `None`); when specified, check if the vertex connectivity of the (di)graph is larger or equal to k. The method thus outputs a boolean only.
- `solver` – string (default: `None`); specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers, see the method `solve`

of the class MixedIntegerLinearProgram. Use method sage.numerical.backends.generic_backend.default_mip_solver() to know which default solver is used or to set the default solver.

- verbose – integer (default: 0); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic application on a PappusGraph:

```
sage: from sage.graphs.connectivity import vertex_connectivity
sage: g=graphs.PappusGraph()
sage: vertex_connectivity(g)
3
sage: g.vertex_connectivity()
3
```

In a grid, the vertex connectivity is equal to the minimum degree, in which case one of the two sets is of cardinality 1:

```
sage: g = graphs.GridGraph([ 3,3 ])
sage: [value, cut, [ setA, setB ]] = vertex_connectivity(g, sets=True)
sage: len(setA) == 1 or len(setB) == 1
True
```

A vertex cut in a tree is any internal vertex:

```
sage: tree = graphs.RandomTree(15)
sage: val, [cut_vertex] = vertex_connectivity(tree, value_only=False)
sage: tree.degree(cut_vertex) > 1
True
```

When value_only = True, this function is optimized for small connectivity values and does not need to build a linear program.

It is the case for connected graphs which are not connected:

```
sage: g = 2 * graphs.PetersenGraph()
sage: vertex_connectivity(g)
0
```

Or if they are just 1-connected:

```
sage: g = graphs.PathGraph(10)
sage: vertex_connectivity(g)
1
```

For directed graphs, the strong connectivity is tested through the dedicated function:

```
sage: g = digraphs.ButterflyGraph(3)
sage: vertex_connectivity(g)
0
```

A complete graph on 10 vertices is 9-connected:

```
sage: g = graphs.CompleteGraph(10)
sage: vertex_connectivity(g)
9
```

A complete digraph on 10 vertices is 9-connected:

```
sage: g = DiGraph(graphs.CompleteGraph(10))
sage: vertex_connectivity(g)
9
```

When parameter `k` is set, we only check for the existence of a vertex cut of order at least `k`:

```
sage: g = graphs.PappusGraph()
sage: vertex_connectivity(g, k=3)
True
sage: vertex_connectivity(g, k=4)
False
```

5.40 Domination

This module implements methods related to the notion of domination in graphs, and more precisely:

dominating_set()	Return a minimum dominating set of the graph.
minimal_dominating_sets()	Return an iterator over the minimal dominating sets of a graph.
is_dominating()	Check whether a set of vertices dominates a graph.
is_redundant()	Check whether a set of vertices has redundant vertices (with respect to domination).
private_neighbors()	Return the private neighbors of a vertex with repect to other vertices.

EXAMPLES:

We compute the size of a minimum dominating set of the Petersen graph:

```
sage: g = graphs.PetersenGraph()
sage: g.dominating_set(value_only=True)
3
```

We enumerate the minimal dominating sets of the 5-star graph:

```
sage: g = graphs.StarGraph(5)
sage: list(g.minimal_dominating_sets())
[{0}, {1, 2, 3, 4, 5}]
```

Now only those that dominate the middle vertex:

```
sage: list(g.minimal_dominating_sets([0]))
[{0}, {1}, {2}, {3}, {4}, {5}]
```

Now the minimal dominating sets of the 5-path graph:

```
sage: g = graphs.PathGraph(5)
sage: list(g.minimal_dominating_sets())
[{0, 2, 4}, {1, 4}, {0, 3}, {1, 3}]
```

We count the minimal dominating sets of the Petersen graph:

```
sage: sum(1 for _ in graphs.PetersenGraph().minimal_dominating_sets())
27
```

5.40.1 Methods

sage.graphs.domination.**dominating_set**(*g*, *independent=False*, *total=False*, *value_only=False*, *solver=None*, *verbose=0*)

Return a minimum dominating set of the graph.

A minimum dominating set S of a graph G is a set of its vertices of minimal cardinality such that any vertex of G is in S or has one of its neighbors in S. See the Wikipedia article Dominating_set.

As an optimization problem, it can be expressed as:

$$\text{Minimize : } \sum_{v \in G} b_v$$

$$\text{Such that : } \forall v \in G, b_v + \sum_{(u,v) \in G.edges()} b_u \geq 1$$

$$\forall x \in G, b_x \text{ is a binary variable}$$

INPUT:

- `independent` – boolean (default: `False`); when `True`, computes a minimum independent dominating set, that is a minimum dominating set that is also an independent set (see also `independent_set()`)
- `total` – boolean (default: `False`); when `True`, computes a total dominating set (see the See the Wikipedia article Dominating_set)
- `value_only` – boolean (default: `False`); whether to only return the cardinality of the computed dominating set, or to return its list of vertices (default)
- `solver` – (default: `None`); specifies a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

A basic illustration on a `PappusGraph`:

```
sage: g = graphs.PappusGraph()
sage: g.dominating_set(value_only=True)
5
```

If we build a graph from two disjoint stars, then link their centers we will find a difference between the cardinality of an independent set and a stable independent set:

```
sage: g = 2 * graphs.StarGraph(5)
sage: g.add_edge(0, 6)
sage: len(g.dominating_set())
2
sage: len(g.dominating_set(independent=True))
6
```

The total dominating set of the Petersen graph has cardinality 4:

```
sage: G = graphs.PetersenGraph()
sage: G.dominating_set(total=True, value_only=True)
4
```

The dominating set is calculated for both the directed and undirected graphs (modification introduced in trac ticket #17905):

```
sage: g = digraphs.Path(3)
sage: g.dominating_set(value_only=True)
2
sage: g = graphs.PathGraph(3)
sage: g.dominating_set(value_only=True)
1
```

sage.graphs.domination.**is_dominating**(*G*, *dom*, *focus=None*)

Check whether `dom` is a dominating set of `G`.

We say that a set D of vertices of a graph G dominates a set S if every vertex of S either belongs to D or is adjacent to a vertex of D. Also, D is a dominating set of G if it dominates $V(G)$.

INPUT:

- `dom` – iterable of vertices of `G`; the vertices of the supposed dominating set.
- `focus` – iterable of vertices of `G` (default: `None`); if specified, this method checks instead if `dom` dominates the vertices in `focus`.

EXAMPLES:

```
sage: g = graphs.CycleGraph(5)
sage: g.is_dominating([0,1], [4, 2])
True

sage: g.is_dominating([0,1])
False
```

sage.graphs.domination.**is_redundant**(*G*, *dom*, *focus=None*)

Check whether `dom` has redundant vertices.

For a graph G and sets D and S of vertices, we say that a vertex $v \in D$ is *redundant* in S if v has no private neighbor with repect to D in S. In other words, there is no vertex in S that is dominated by v but not by $D \setminus \{v\}$.

INPUT:

- `dom` – iterable of vertices of `G`; where we look for redundant vertices.
- `focus` – iterable of vertices of `G` (default: `None`); if specified, this method checks instead whether `dom` has a redundant vertex in `focus`.

Warning: The assumption is made that `focus` (if provided) does not contain repeated vertices.

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: G.is_redundant(['000', '101'], ['011'])
True
sage: G.is_redundant(['000', '101'])
False
```

sage.graphs.domination.**minimal_dominating_sets**(*G*, *to_dominate=None*, *work_on_copy=True*)

Return an iterator over the minimal dominating sets of a graph.

INPUT:

- `G` – a graph.

- `to_dominate` – vertex iterable or `None` (default: `None`); the set of vertices to be dominated.
- `work_on_copy` – boolean (default: `True`); whether or not to work on a copy of the input graph; if set to `False`, the input graph will be modified (relabeled).

OUTPUT:

An iterator over the inclusion-minimal sets of vertices of `G`. If `to_dominate` is provided, return an iterator over the inclusion-minimal sets of vertices that dominate the vertices of `to_dominate`.

ALGORITHM: The algorithm described in [BDHPR2019].

AUTHOR: Jean-Florent Raymond (2019-03-04) – initial version.

EXAMPLES:

```
sage: G = graphs.ButterflyGraph()
sage: ll = list(G.minimal_dominating_sets())
sage: pp = [{0, 1}, {1, 3}, {0, 2}, {2, 3}, {4}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x in pp)
True

sage: ll = list(G.minimal_dominating_sets([0,3]))
sage: pp = [{0}, {3}, {4}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x in pp)
True

sage: ll = list(G.minimal_dominating_sets([4]))
sage: pp = [{4}, {0}, {1}, {2}, {3}]
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x in pp)
True
```

```
sage: ll = list(graphs.PetersenGraph().minimal_dominating_sets())
sage: pp = [{0, 2, 6},
....: {0, 9, 3},
....: {0, 8, 7},
....: {1, 3, 7},
....: {1, 4, 5},
....: {8, 1, 9},
....: {8, 2, 4},
....: {9, 2, 5},
....: {3, 5, 6},
....: {4, 6, 7},
....: {0, 8, 2, 9},
....: {0, 3, 6, 7},
....: {1, 3, 5, 9},
....: {8, 1, 4, 7},
....: {2, 4, 5, 6},
....: {0, 1, 2, 3, 4},
....: {0, 1, 2, 5, 7},
....: {0, 1, 4, 6, 9},
....: {0, 1, 5, 6, 8},
....: {0, 8, 3, 4, 5},
....: {0, 9, 4, 5, 7},
....: {8, 1, 2, 3, 6},
....: {1, 2, 9, 6, 7},
....: {9, 2, 3, 4, 7},
....: {8, 2, 3, 5, 7},
....: {8, 9, 3, 4, 6},
....: {8, 9, 5, 6, 7}]
```

(continues on next page)

(continued from previous page)

```
sage: len(ll) == len(pp) and all(x in pp for x in ll) and all(x in ll for x in pp)
True
```

sage.graphs.domination.**private_neighbors**(*G*, *vertex*, *dom*)

Return the private neighbors of a vertex with repect to other vertices.

A private neighbor of a vertex v with respect to a vertex subset D is a closed neighbor of v that is not dominated by a vertex of $D \setminus \{v\}$.

INPUT:

- `vertex` – a vertex of `G`.
- `dom` – iterable of vertices of `G`; the vertices possibly stealing private neighbors from `vertex`.

OUTPUT:

Return the closed neighbors of `vertex` that are not closed neighbors of any other vertex of `dom`.

EXAMPLES:

```
sage: g = graphs.PathGraph(5)
sage: list(g.private_neighbors(1, [1, 3, 4]))
[1, 0]

sage: list(g.private_neighbors(1, [3, 4]))
[1, 0]

sage: list(g.private_neighbors(1, [3, 4, 0]))
[]
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[Lein] Tom Leinster, *The magnitude of metric spaces*. Doc. Math. 18 (2013), 857-905.

PYTHON MODULE INDEX

c

sage.combinat.designs.incidence_structures, 683

g

sage.graphs.asteroidal_triples, 716
sage.graphs.base.boost_graph, 670
sage.graphs.base.c_graph, 602
sage.graphs.base.dense_graph, 641
sage.graphs.base.graph_backends, 664
sage.graphs.base.overview, 601
sage.graphs.base.sparse_graph, 630
sage.graphs.base.static_dense_graph, 649
sage.graphs.base.static_sparse_backend, 658
sage.graphs.base.static_sparse_graph, 652
sage.graphs.bipartite_graph, 383
sage.graphs.centrality, 714
sage.graphs.cliquer, 712
sage.graphs.comparability, 720
sage.graphs.connectivity, 930
sage.graphs.convexity_properties, 862
sage.graphs.digraph, 342
sage.graphs.digraph_generators, 542
sage.graphs.distances_all_pairs, 868
sage.graphs.domination, 955
sage.graphs.generic_graph, 1
sage.graphs.generic_graph_pyx, 923
sage.graphs.genus, 748
sage.graphs.graph, 228
sage.graphs.graph_coloring, 701
sage.graphs.graph_database, 555
sage.graphs.graph_decompositions.bandwidth, 815
sage.graphs.graph_decompositions.clique_separators, 858
sage.graphs.graph_decompositions.cutwidth, 817
sage.graphs.graph_decompositions.graph_products, 822
sage.graphs.graph_decompositions.modular_decomposition, 825
sage.graphs.graph_decompositions.rankwidth, 812
sage.graphs.graph_decompositions.vertex_separation, 800

sage.graphs.graph_editor, 890
sage.graphs.graph_generators, 403
sage.graphs.graph_generators_pyx, 555
sage.graphs.graph_input, 894
sage.graphs.graph_latex, 876
sage.graphs.graph_list, 890
sage.graphs.graph_plot, 766
sage.graphs.graph_plot_js, 797
sage.graphs.hyperbolicity, 897
sage.graphs.hypergraph_generators, 681
sage.graphs.independent_sets, 718
sage.graphs.isgci, 591
sage.graphs.line_graph, 726
sage.graphs.lovasz_theta, 751
sage.graphs.matchpoly, 745
sage.graphs.orientations, 929
sage.graphs.partial_cube, 907
sage.graphs.path_enumeration, 909
sage.graphs.planarity, 754
sage.graphs.pq_trees, 738
sage.graphs.schnyder, 751
sage.graphs.spanning_tree, 730
sage.graphs.strongly_regular_db, 569
sage.graphs.traversals, 755
sage.graphs.trees, 744
sage.graphs.tutte_polynomial, 902
sage.graphs.views, 396
sage.graphs.weakly_chordal, 865

INDEX

Symbols

__eq__() (*sage.graphs.generic_graph.GenericGraph method*), 7

A

acyclic_edge_coloring() (*in module sage.graphs.graph_coloring*), 702
add_arc() (*sage.graphs.base.c_graph.CGraph method*), 603
add_arc_label() (*sage.graphs.base.sparse_graph.SparseGraph method*), 634
add_clique() (*sage.graphs.generic_graph.GenericGraph method*), 8
add_cycle() (*sage.graphs.generic_graph.GenericGraph method*), 8
add_edge() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 645
add_edge() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 665
add_edge() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 638
add_edge() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
add_edge() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 387
add_edge() (*sage.graphs.generic_graph.GenericGraph method*), 9
add_edges() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 645
add_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 665
add_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 638
add_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
add_edges() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 387
add_edges() (*sage.graphs.generic_graph.GenericGraph method*), 10
add_path() (*sage.graphs.generic_graph.GenericGraph method*), 10
add_vertex() (*sage.graphs.base.c_graph.CGraph method*), 603
add_vertex() (*sage.graphs.base.c_graph.CGraphBackend method*), 615
add_vertex() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 666
add_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
add_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 663
add_vertex() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 388
add_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 11
add_vertices() (*sage.graphs.base.c_graph.CGraph method*), 605
add_vertices() (*sage.graphs.base.c_graph.CGraphBackend method*), 616
add_vertices() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 666
add_vertices() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
add_vertices() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 388
add_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 11
adjacency_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 12
AffineOrthogonalPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 410

AfricaMap() (*sage.graphs.graph_generators.GraphGenerators static method*), 411
AhrensSzekeresGeneralizedQuadrangleGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 411
all_arcs() (*sage.graphs.base.c_graph.CGraph method*), 605
all_arcs() (*sage.graphs.base.sparse_graph.SparseGraph method*), 634
all_cliques() (*in module sage.graphs.cliquer*), 712
all_cliques() (*sage.graphs.graph.Graph method*), 244
all_cycles_iterator() (*sage.graphs.digraph.DiGraph method*), 348
all_graph_colorings() (*in module sage.graphs.graph_coloring*), 703
all_max_clique() (*in module sage.graphs.cliquer*), 713
all_paths() (*in module sage.graphs.path_enumeration*), 910
all_paths() (*sage.graphs.generic_graph.GenericGraph method*), 13
all_paths_iterator() (*in module sage.graphs.path_enumeration*), 912
all_paths_iterator() (*sage.graphs.digraph.DiGraph method*), 349
all_simple_cycles() (*sage.graphs.digraph.DiGraph method*), 352
all_simple_paths() (*in module sage.graphs.path_enumeration*), 915
all_simple_paths() (*sage.graphs.digraph.DiGraph method*), 355
allow_loops() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 389
allow_loops() (*sage.graphs.generic_graph.GenericGraph method*), 15
allow_multiple_edges() (*sage.graphs.generic_graph.GenericGraph method*), 16
allows_loops() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
allows_loops() (*sage.graphs.generic_graph.GenericGraph method*), 17
allows_multiple_edges() (*sage.graphs.generic_graph.GenericGraph method*), 18
am() (*sage.graphs.generic_graph.GenericGraph method*), 19
antisymmetric() (*sage.graphs.generic_graph.GenericGraph method*), 20
apex_vertices() (*sage.graphs.graph.Graph method*), 245
apparently_feasible_parameters() (*in module sage.graphs.strongly_regular_db*), 575
append_child() (*sage.graphs.schnyder.TreeNode method*), 752
arboricity() (*sage.graphs.graph.Graph method*), 246
arc_label() (*sage.graphs.base.sparse_graph.SparseGraph method*), 634
assembly() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 827
atoms_and_clique_separators() (*in module sage.graphs.graph_decompositions.clique_separators*), 858
atoms_and_clique_separators() (*sage.graphs.graph.Graph method*), 247
automorphism_group() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 686
automorphism_group() (*sage.graphs.generic_graph.GenericGraph method*), 20
average_degree() (*sage.graphs.generic_graph.GenericGraph method*), 23
average_distance() (*sage.graphs.generic_graph.GenericGraph method*), 23
AztecDiamondGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 412

B

b_coloring() (*in module sage.graphs.graph_coloring*), 704
Balaban10Cage() (*sage.graphs.graph_generators.GraphGenerators static method*), 412
Balaban11Cage() (*sage.graphs.graph_generators.GraphGenerators static method*), 413
BalancedTree() (*sage.graphs.graph_generators.GraphGenerators static method*), 414
bandwidth() (*in module sage.graphs.graph_decompositions.bandwidth*), 816
bandwidth_heuristics() (*in module sage.graphs.base.boost_graph*), 671
BarbellGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 415
BidiakisCube() (*sage.graphs.graph_generators.GraphGenerators static method*), 415
bidirectional_dijkstra() (*sage.graphs.base.c_graph.CGraphBackend method*), 617
bidirectional_dijkstra_special() (*sage.graphs.base.c_graph.CGraphBackend method*), 617

BiggsSmithGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 416
binary_string_from_dig6() (*in module sage.graphs.generic_graph_pyx*), 923
binary_string_from_graph6() (*in module sage.graphs.generic_graph_pyx*), 924
binary_string_to_graph6() (*in module sage.graphs.generic_graph_pyx*), 924
BinomialRandomUniform() (*sage.graphs.hypergraph_generators.HypergraphGenerators method*), 681
bipartite_color() (*sage.graphs.graph.Graph method*), 251
bipartite_sets() (*sage.graphs.graph.Graph method*), 251
BipartiteGraph (*class in sage.graphs.bipartite_graph*), 383
bipartition() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 389
BishopGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 417
BlanusaFirstSnarkGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 417
BlanusaSecondSnarkGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 418
block_sizes() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 687
blocks() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 687
blocks_and_cut_vertices() (*in module sage.graphs.base.boost_graph*), 671
blocks_and_cut_vertices() (*in module sage.graphs.connectivity*), 936
blocks_and_cut_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 24
blocks_and_cuts_tree() (*in module sage.graphs.connectivity*), 937
blocks_and_cuts_tree() (*sage.graphs.generic_graph.GenericGraph method*), 25
boruvka() (*in module sage.graphs.spanning_tree*), 731
bounded_outdegree_orientation() (*sage.graphs.graph.Graph method*), 251
breadth_first_level_search() (*in module sage.graphs.partial_cube*), 908
breadth_first_search() (*sage.graphs.base.c_graph.CGraphBackend method*), 618
breadth_first_search() (*sage.graphs.generic_graph.GenericGraph method*), 26
bridges() (*in module sage.graphs.connectivity*), 938
bridges() (*sage.graphs.graph.Graph method*), 252
BrinkmannGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 418
BrouwerHaemersGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 419
BubbleSortGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 419
BuckyBall() (*sage.graphs.graph_generators.GraphGenerators static method*), 420
BullGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 420
ButterflyGraph() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 544
ButterflyGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 421

C

c_graph() (*sage.graphs.base.c_graph.CGraphBackend method*), 619
CaiFurerImmermanGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 422
CameronGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 423
canaug_traverse_edge() (*in module sage.graphs.graph_generators*), 540
canaug_traverse_vert() (*in module sage.graphs.graph_generators*), 540
canonical_label() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 687
canonical_label() (*sage.graphs.generic_graph.GenericGraph method*), 28
cardinality() (*sage.graphs.generic_graph_pyx.SubgraphSearch method*), 923
cardinality() (*sage.graphs.independent_sets.IndependentSets method*), 719
cardinality() (*sage.graphs.pq_trees.P method*), 739
cardinality() (*sage.graphs.pq_trees.Q method*), 742
cartesian_product() (*sage.graphs.generic_graph.GenericGraph method*), 30
categorical_product() (*sage.graphs.generic_graph.GenericGraph method*), 30
Cell120() (*sage.graphs.graph_generators.GraphGenerators static method*), 424
Cell600() (*sage.graphs.graph_generators.GraphGenerators static method*), 424

center() (*sage.graphs.generic_graph.GenericGraph method*), 30
centrality_betweenness() (*in module sage.graphs.centrality*), 714
centrality_betweenness() (*sage.graphs.generic_graph.GenericGraph method*), 31
centrality_closeness() (*sage.graphs.generic_graph.GenericGraph method*), 32
centrality_closeness_random_k() (*in module sage.graphs.centrality*), 715
centrality_closeness_top_k() (*in module sage.graphs.centrality*), 715
centrality_degree() (*sage.graphs.graph.Graph method*), 252
CGraph (*class in sage.graphs.base.c_graph*), 602
CGraphBackend (*class in sage.graphs.base.c_graph*), 615
chang_graphs() (*sage.graphs.graph_generators.GraphGenerators static method*), 527
characteristic_polynomial() (*sage.graphs.generic_graph.GenericGraph method*), 34
charpoly() (*sage.graphs.generic_graph.GenericGraph method*), 35
check_aut() (*in module sage.graphs.graph_generators*), 541
check_aut_edge() (*in module sage.graphs.graph_generators*), 541
check_parallel() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 828
check_prime() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 830
check_series() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 831
check_tkz_graph() (*in module sage.graphs.graph_latex*), 889
check_vertex() (*sage.graphs.base.c_graph.CGraph method*), 606
ChessboardGraphGenerator() (*sage.graphs.graph_generators.GraphGenerators static method*), 424
children_node_type() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 833
chromatic_index() (*sage.graphs.graph.Graph method*), 253
chromatic_number() (*in module sage.graphs.graph_coloring*), 705
chromatic_number() (*sage.graphs.graph.Graph method*), 254
chromatic_polynomial() (*sage.graphs.graph.Graph method*), 255
chromatic_quasisymmetric_function() (*sage.graphs.graph.Graph method*), 256
chromatic_symmetric_function() (*sage.graphs.graph.Graph method*), 257
ChvatalGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 425
Circuit() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 545
Circulant() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 546
CirculantGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 425
CircularLadderGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 427
classes() (*sage.graphs.isgci.GraphClasses method*), 598
ClawGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 427
clear() (*sage.graphs.generic_graph.GenericGraph method*), 35
clear_node_split_info() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 833
cleave() (*in module sage.graphs.connectivity*), 938
cleave() (*sage.graphs.graph.Graph method*), 258
ClebschGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 428
clique_complex() (*sage.graphs.graph.Graph method*), 259
clique_maximum() (*sage.graphs.graph.Graph method*), 260
clique_number() (*in module sage.graphs.cliquer*), 713
clique_number() (*sage.graphs.graph.Graph method*), 260
clique_polynomial() (*sage.graphs.graph.Graph method*), 262
cliques_containing_vertex() (*sage.graphs.graph.Graph method*), 262
cliques_get_clique_bipartite() (*sage.graphs.graph.Graph method*), 263
cliques_get_max_clique_graph() (*sage.graphs.graph.Graph method*), 263
cliques_maximal() (*sage.graphs.graph.Graph method*), 263
cliques_maximum() (*sage.graphs.graph.Graph method*), 264
cliques_number_of() (*sage.graphs.graph.Graph method*), 265

cliques_vertex_clique_number() (*sage.graphs.graph.Graph method*), 265
cluster_transitivity() (*sage.graphs.generic_graph.GenericGraph method*), 36
cluster_triangles() (*sage.graphs.generic_graph.GenericGraph method*), 36
clustering_average() (*sage.graphs.generic_graph.GenericGraph method*), 37
clustering_coeff() (*in module sage.graphs.base.boost_graph*), 672
clustering_coeff() (*sage.graphs.generic_graph.GenericGraph method*), 37
coarsest_equitable_refinement() (*sage.graphs.generic_graph.GenericGraph method*), 38
coloring() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 687
coloring() (*sage.graphs.graph.Graph method*), 266
common_neighbors_matrix() (*sage.graphs.graph.Graph method*), 267
complement() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 688
complement() (*sage.graphs.base.dense_graph.DenseGraph method*), 643
complement() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 389
complement() (*sage.graphs.generic_graph.GenericGraph method*), 39
Complete() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 546
complete_poly() (*in module sage.graphs.matchpoly*), 745
CompleteBipartiteGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 428
CompleteGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 430
CompleteMultipartiteGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 431
CompleteUniform() (*sage.graphs.hypergraph_generators.HypergraphGenerators method*), 682
compute_depth_of_self_and_children() (*sage.graphs.schnyder.TreeNode method*), 752
compute_mu_for_co_component() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 834
compute_mu_for_component() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 835
compute_number_of_descendants() (*sage.graphs.schnyder.TreeNode method*), 753
connected_component_containing_vertex() (*in module sage.graphs.connectivity*), 940
connected_component_containing_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 39
connected_components() (*in module sage.graphs.connectivity*), 940
connected_components() (*sage.graphs.generic_graph.GenericGraph method*), 40
connected_components_number() (*in module sage.graphs.connectivity*), 941
connected_components_number() (*sage.graphs.generic_graph.GenericGraph method*), 40
connected_components_sizes() (*in module sage.graphs.connectivity*), 941
connected_components_sizes() (*sage.graphs.generic_graph.GenericGraph method*), 40
connected_components_subgraphs() (*in module sage.graphs.connectivity*), 941
connected_components_subgraphs() (*sage.graphs.generic_graph.GenericGraph method*), 41
connected_subgraph_iterator() (*in module sage.graphs.base.static_dense_graph*), 649
connected_subgraph_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 41
contract_edge() (*sage.graphs.generic_graph.GenericGraph method*), 42
contract_edges() (*sage.graphs.generic_graph.GenericGraph method*), 43
contracted_edge() (*in module sage.graphs.tutte_polynomial*), 904
convexity_properties() (*sage.graphs.graph.Graph method*), 268
ConvexityProperties (*class in sage.graphs.convexity_properties*), 863
copy() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 689
copy() (*sage.graphs.generic_graph.GenericGraph method*), 43
cores() (*sage.graphs.graph.Graph method*), 269
cospectral_graphs() (*sage.graphs.graph_generators.GraphGenerators method*), 527
CossidentePenttilaGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 431
CoxeterGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 432
create_normal_node() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 836
create_parallel_node() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 836

create_prime_node() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 836
create_series_node() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 837
crossing_number() (*sage.graphs.generic_graph.GenericGraph method*), 45
CubeConnectedCycle() (*sage.graphs.graph_generators.GraphGenerators static method*), 432
CubeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 433
current_allocation() (*sage.graphs.base.c_graph.CGraph method*), 607
cutwidth() (*in module sage.graphs.graph_decompositions.cutwidth*), 819
cutwidth_dyn() (*in module sage.graphs.graph_decompositions.cutwidth*), 821
cutwidth_MILP() (*in module sage.graphs.graph_decompositions.cutwidth*), 820
cycle_basis() (*sage.graphs.generic_graph.GenericGraph method*), 45
CycleGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 433

D

data_to_degseq() (*in module sage.graphs.graph_database*), 567
DeBruijn() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 546
degree() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 689
degree() (*sage.graphs.base.c_graph.CGraphBackend method*), 619
degree() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 666
degree() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 660
degree() (*sage.graphs.generic_graph.GenericGraph method*), 47
degree_constrained_subgraph() (*sage.graphs.graph.Graph method*), 270
degree_histogram() (*sage.graphs.generic_graph.GenericGraph method*), 48
degree_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 48
degree_polynomial() (*sage.graphs.digraph.DiGraph method*), 357
degree_sequence() (*sage.graphs.generic_graph.GenericGraph method*), 50
degree_to_cell() (*sage.graphs.generic_graph.GenericGraph method*), 51
degrees() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 689
DegreeSequence() (*sage.graphs.graph_generators.GraphGenerators static method*), 434
DegreeSequenceBipartite() (*sage.graphs.graph_generators.GraphGenerators static method*), 435
DegreeSequenceConfigurationModel() (*sage.graphs.graph_generators.GraphGenerators static method*), 435
DegreeSequenceExpected() (*sage.graphs.graph_generators.GraphGenerators static method*), 436
DegreeSequenceTree() (*sage.graphs.graph_generators.GraphGenerators static method*), 436
degseq_to_data() (*in module sage.graphs.graph_database*), 567
DejterGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 436
del_all_arcs() (*sage.graphs.base.c_graph.CGraph method*), 608
del_arc_label() (*sage.graphs.base.sparse_graph.SparseGraph method*), 635
del_edge() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 646
del_edge() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 666
del_edge() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 638
del_edge() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
del_vertex() (*sage.graphs.base.c_graph.CGraph method*), 608
del_vertex() (*sage.graphs.base.c_graph.CGraphBackend method*), 620
del_vertex() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 666
del_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
del_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 663
del_vertices() (*sage.graphs.base.c_graph.CGraphBackend method*), 620
del_vertices() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
delete_edge() (*sage.graphs.generic_graph.GenericGraph method*), 51
delete_edges() (*sage.graphs.generic_graph.GenericGraph method*), 52

delete_multiedge() (*sage.graphs.generic_graph.GenericGraph method*), 52
delete_vertex() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 389
delete_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 53
delete_vertices() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 390
delete_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 53
DenseGraph (*class in sage.graphs.base.dense_graph*), 643
DenseGraphBackend (*class in sage.graphs.base.dense_graph*), 644
density() (*sage.graphs.generic_graph.GenericGraph method*), 54
depth_first_search() (*sage.graphs.base.c_graph.CGraphBackend method*), 621
depth_first_search() (*sage.graphs.generic_graph.GenericGraph method*), 54
depth_first_traversal() (*in module sage.graphs.partial_cube*), 908
DesarguesGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 437
description() (*sage.graphs.isgci.GraphClass method*), 597
diameter() (*in module sage.graphs.distances_all_pairs*), 869
diameter() (*sage.graphs.generic_graph.GenericGraph method*), 55
DiamondGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 437
dig6_string() (*sage.graphs.digraph.DiGraph method*), 357
DiGraph (*class in sage.graphs.digraph*), 343
DiGraphGenerators (*class in sage.graphs.digraph_generators*), 543
DipoleGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 437
disjoint_routed_paths() (*sage.graphs.generic_graph.GenericGraph method*), 56
disjoint_union() (*sage.graphs.generic_graph.GenericGraph method*), 57
disjunctive_product() (*sage.graphs.generic_graph.GenericGraph method*), 58
distance() (*sage.graphs.generic_graph.GenericGraph method*), 58
distance_all_pairs() (*sage.graphs.generic_graph.GenericGraph method*), 59
distance_graph() (*sage.graphs.generic_graph.GenericGraph method*), 60
distance_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 61
distances_all_pairs() (*in module sage.graphs.distances_all_pairs*), 870
distances_and_predecessors_all_pairs() (*in module sage.graphs.distances_all_pairs*), 871
distances_distribution() (*in module sage.graphs.distances_all_pairs*), 871
distances_distribution() (*sage.graphs.generic_graph.GenericGraph method*), 62
DodecahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 437
dominating_set() (*in module sage.graphs.domination*), 956
dominating_set() (*sage.graphs.generic_graph.GenericGraph method*), 63
dominator_tree() (*in module sage.graphs.base.boost_graph*), 673
dominator_tree() (*sage.graphs.generic_graph.GenericGraph method*), 64
DorogovtsevGoltsevMendesGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 438
dot2tex_picture() (*sage.graphs.graph_latex.GraphLatex method*), 882
DoubleStarSnark() (*sage.graphs.graph_generators.GraphGenerators static method*), 438
dual() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 690
DurerGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 438
DyckGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 439

E

Ear (*class in sage.graphs.tutte_polynomial*), 903
ear_decomposition() (*sage.graphs.graph.Graph method*), 271
eccentricity() (*in module sage.graphs.distances_all_pairs*), 872
eccentricity() (*sage.graphs.generic_graph.GenericGraph method*), 66
edge_boundary() (*sage.graphs.generic_graph.GenericGraph method*), 67
edge_coloring() (*in module sage.graphs.graph_coloring*), 705

edge_coloring() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 690
edge_connectivity() (*in module sage.graphs.base.boost_graph*), 674
edge_connectivity() (*in module sage.graphs.connectivity*), 942
edge_connectivity() (*sage.graphs.generic_graph.GenericGraph method*), 68
edge_cut() (*sage.graphs.generic_graph.GenericGraph method*), 70
edge_disjoint_paths() (*sage.graphs.generic_graph.GenericGraph method*), 72
edge_disjoint_spanning_trees() (*sage.graphs.generic_graph.GenericGraph method*), 72
edge_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 73
edge_label() (*sage.graphs.generic_graph.GenericGraph method*), 74
edge_labels() (*sage.graphs.generic_graph.GenericGraph method*), 74
edge_multiplicities() (*in module sage.graphs.tutte_polynomial*), 905
edges() (*sage.graphs.generic_graph.GenericGraph method*), 75
edges_incident() (*sage.graphs.generic_graph.GenericGraph method*), 76
EdgeSelection (*class in sage.graphs.tutte_polynomial*), 904
EdgesView (*class in sage.graphs.views*), 396
effective_resistance() (*sage.graphs.graph.Graph method*), 272
effective_resistance_matrix() (*sage.graphs.graph.Graph method*), 273
EgawaGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 440
eigenmatrix() (*in module sage.graphs.strongly_regular_db*), 576
eigenspaces() (*sage.graphs.generic_graph.GenericGraph method*), 76
eigenvectors() (*sage.graphs.generic_graph.GenericGraph method*), 78
either_connected_or_not_connected() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 837
EllinghamHorton54Graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 441
EllinghamHorton78Graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 441
EmptyGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 442
equivalent_trees() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 837
ErreraGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 442
eulerian_circuit() (*sage.graphs.generic_graph.GenericGraph method*), 79
eulerian_orientation() (*sage.graphs.generic_graph.GenericGraph method*), 80
EuropeMap() (*sage.graphs.graph_generators.GraphGenerators static method*), 443
export_to_file() (*sage.graphs.generic_graph.GenericGraph method*), 81

F

F26AGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 443
faces() (*sage.graphs.generic_graph.GenericGraph method*), 82
feedback_edge_set() (*sage.graphs.digraph.DiGraph method*), 357
feedback_vertex_set() (*sage.graphs.generic_graph.GenericGraph method*), 83
feng_k_shortest_simple_paths() (*in module sage.graphs.path_enumeration*), 917
FibonacciTree() (*sage.graphs.graph_generators.GraphGenerators static method*), 444
filter_kruskal() (*in module sage.graphs.spanning_tree*), 732
filter_kruskal_iterator() (*in module sage.graphs.spanning_tree*), 733
find_ear() (*sage.graphs.tutte_polynomial.Ear static method*), 903
find_hamiltonian() (*in module sage.graphs.generic_graph_pyx*), 925
first_coloring() (*in module sage.graphs.graph_coloring*), 707
flatten() (*in module sage.graphs.pq_trees*), 743
flatten() (*sage.graphs.pq_trees.PQ method*), 740
flow() (*sage.graphs.generic_graph.GenericGraph method*), 85
flow_polytope() (*sage.graphs.digraph.DiGraph method*), 359
FlowerSnark() (*sage.graphs.graph_generators.GraphGenerators static method*), 444

floyd_warshall() (*in module sage.graphs.distances_all_pairs*), 873
floyd_warshall_shortest_paths() (*in module sage.graphs.base.boost_graph*), 674
FoldedCubeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 444
FolkmanGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 445
forbidden_subgraphs() (*sage.graphs.isgci.GraphClass method*), 597
form_module() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 838
FosterGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 445
fractional_chromatic_index() (*sage.graphs.graph.Graph method*), 274
FranklinGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 445
FriendshipGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 446
from_adjacency_matrix() (*in module sage.graphs.graph_input*), 894
from_dict_of_dicts() (*in module sage.graphs.graph_input*), 894
from_dict_of_lists() (*in module sage.graphs.graph_input*), 895
from_dig6() (*in module sage.graphs.graph_input*), 895
from_graph6() (*in module sage.graphs.graph_input*), 895
from_graph6() (*in module sage.graphs.graph_list*), 891
from_incidence_matrix() (*in module sage.graphs.graph_input*), 896
from_oriented_incidence_matrix() (*in module sage.graphs.graph_input*), 896
from_seidel_adjacency_matrix() (*in module sage.graphs.graph_input*), 896
from_sparse6() (*in module sage.graphs.graph_input*), 897
from_sparse6() (*in module sage.graphs.graph_list*), 891
from_whatever() (*in module sage.graphs.graph_list*), 891
FruchtGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 447
fullerenes() (*sage.graphs.graph_generators.GraphGenerators method*), 529
FurerGadget() (*sage.graphs.graph_generators.GraphGenerators static method*), 448
fusenes() (*sage.graphs.graph_generators.GraphGenerators method*), 530
FuzzyBallGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 449

G

gamma_classes() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 838
gen_html_code() (*in module sage.graphs.graph_plot_js*), 798
GeneralizedDeBruijn() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 547
GeneralizedPetersenGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 449
GenericGraph (*class in sage.graphs.generic_graph*), 7
GenericGraph_pyx (*class in sage.graphs.generic_graph_pyx*), 923
GenericGraphBackend (*class in sage.graphs.base.graph_backends*), 665
GenericGraphQuery (*class in sage.graphs.graph_database*), 556
genus() (*sage.graphs.generic_graph.GenericGraph method*), 86
genus() (*sage.graphs.genus.simple_connected_genus_backtracker method*), 749
get_child_splits() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 839
get_class() (*sage.graphs.isgci.GraphClasses method*), 598
get_edge_label() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 646
get_edge_label() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
get_edge_label() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 639
get_edge_label() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
get_embedding() (*sage.graphs.generic_graph.GenericGraph method*), 88
get_embedding() (*sage.graphs.genus.simple_connected_genus_backtracker method*), 749
get_graphs_list() (*sage.graphs.graph_database.GraphQuery method*), 564
get_module_type() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 839
get_option() (*sage.graphs.graph_latex.GraphLatex method*), 883

get_pos() (*sage.graphs.generic_graph.GenericGraph method*), 89
get_spqr_tree() (*sage.graphs.connectivity.TriconnectivitySPQR method*), 933
get_triconnected_components() (*sage.graphs.connectivity.TriconnectivitySPQR method*), 935
get_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 89
get_vertex_in() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 839
get_vertices() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 840
get_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 89
girth() (*sage.graphs.generic_graph.GenericGraph method*), 90
GoethalsSeidelGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 450
GoldnerHararyGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 450
GolombGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 451
gomory_hu_tree() (*sage.graphs.graph.Graph method*), 275
GossetGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 451
Graph (*class in sage.graphs.graph*), 237
graph6_string() (*sage.graphs.graph.Graph method*), 276
graph6_to_plot() (*in module sage.graphs.graph_database*), 567
graph_classes (*in module sage.graphs.isgci*), 600
graph_db_info() (*in module sage.graphs.graph_database*), 568
graph_editor() (*in module sage.graphs.graph_editor*), 890
graph_isom_equivalent_non_edge_labeled_graph() (*in module sage.graphs.generic_graph*), 225
graph_to_js() (*in module sage.graphs.graph_editor*), 890
GraphClass (*class in sage.graphs.isgci*), 596
GraphClasses (*class in sage.graphs.isgci*), 597
GraphDatabase (*class in sage.graphs.graph_database*), 557
GraphGenerators (*class in sage.graphs.graph_generators*), 407
GraphLatex (*class in sage.graphs.graph_latex*), 881
GraphPlot (*class in sage.graphs.graph_plot*), 771
graphplot() (*sage.graphs.generic_graph.GenericGraph method*), 90
GraphQuery (*class in sage.graphs.graph_database*), 563
graphviz_string() (*sage.graphs.generic_graph.GenericGraph method*), 91
graphviz_to_file_named() (*sage.graphs.generic_graph.GenericGraph method*), 96
GrayGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 452
greedy_is_comparability() (*in module sage.graphs.comparability*), 722
greedy_is_comparability_with_certificate() (*in module sage.graphs.comparability*), 723
Grid2dGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 452
GridGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 452
GrotzschGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 453
ground_set() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 690
grundy_coloring() (*in module sage.graphs.graph_coloring*), 707

H

habib_maurer_algorithm() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 840
HaemersGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 453
HallJankoGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 454
hamiltonian_cycle() (*sage.graphs.generic_graph.GenericGraph method*), 96
hamiltonian_path() (*sage.graphs.generic_graph.GenericGraph method*), 98
HammingGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 455
HanoiTowerGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 456
HararyGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 457
HarborthGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 458

HarriesGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 458
HarriesWongGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 458
has_arc() (*sage.graphs.base.c_graph.CGraph method*), 610
has_arc() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 663
has_arc_label() (*sage.graphs.base.sparse_graph.SparseGraph method*), 635
has_edge() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 647
has_edge() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
has_edge() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 639
has_edge() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
has_edge() (*sage.graphs.generic_graph.GenericGraph method*), 99
has_homomorphism_to() (*sage.graphs.graph.Graph method*), 276
has_left_cocomponent_fragment() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 842
has_left_split() (*sage.graphs.graph_decompositions.modular_decomposition.Node method*), 826
has_loops() (*sage.graphs.generic_graph.GenericGraph method*), 99
has_multiple_edges() (*sage.graphs.generic_graph.GenericGraph method*), 100
has_perfect_matching() (*sage.graphs.graph.Graph method*), 277
has_right_component_fragment() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 843
has_right_layer_neighbor() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 844
has_right_split() (*sage.graphs.graph_decompositions.modular_decomposition.Node method*), 826
has_vertex() (*sage.graphs.base.c_graph.CGraph method*), 610
has_vertex() (*sage.graphs.base.c_graph.CGraphBackend method*), 622
has_vertex() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
has_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
has_vertex() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664
has_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 101
have_tkz_graph() (*in module sage.graphs.graph_latex*), 889
HeawoodGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 459
HerschelGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 460
HexahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 460
HigmanSimsGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 461
HoffmanGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 462
HoffmanSingletonGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 462
HoltGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 463
HortonGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 463
HouseGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 464
HouseXGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 464
hull() (*sage.graphs.convexity_properties.ConvexityProperties method*), 864
hull_number() (*sage.graphs.convexity_properties.ConvexityProperties method*), 864
hyperbolicity() (*in module sage.graphs.hyperbolicity*), 899
hyperbolicity_distribution() (*in module sage.graphs.hyperbolicity*), 901
HypergraphGenerators (*class in sage.graphs.hypergraph_generators*), 681
HyperStarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 464

I

IcosahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 465
igraph_feature() (*in module sage.graphs.generic_graph*), 227
igraph_graph() (*sage.graphs.generic_graph.GenericGraph method*), 101
ihara_zeta_function_inverse() (*sage.graphs.graph.Graph method*), 278

ImaseItoh() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 548
in_branchings() (*sage.graphs.digraph.DiGraph method*), 361
in_degree() (*sage.graphs.base.c_graph.CGraphBackend method*), 622
in_degree() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
in_degree() (*sage.graphs.base.sparse_graph.SparseGraph method*), 636
in_degree() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 661
in_degree() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664
in_degree() (*sage.graphs.digraph.DiGraph method*), 363
in_degree_iterator() (*sage.graphs.digraph.DiGraph method*), 363
in_degree_sequence() (*sage.graphs.digraph.DiGraph method*), 363
in_neighbors() (*sage.graphs.base.c_graph.CGraph method*), 611
in_neighbors() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664
incidence_graph() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 691
incidence_matrix() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 691
incidence_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 103
IncidenceStructure (*class in sage.combinat.designs.incidence_structures*), 685
inclusion_digraph() (*sage.graphs.isgci.GraphClasses method*), 598
inclusions() (*sage.graphs.isgci.GraphClasses method*), 598
incoming_edge_iterator() (*sage.graphs.digraph.DiGraph method*), 364
incoming_edges() (*sage.graphs.digraph.DiGraph method*), 364
independent_set() (*sage.graphs.graph.Graph method*), 278
independent_set_of_representatives() (*sage.graphs.graph.Graph method*), 279
IndependentSets (*class in sage.graphs.independent_sets*), 718
induced_substructure() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 691
int_to_binary_string() (*in module sage.graphs.generic_graph_pyx*), 926
interactive_query() (*sage.graphs.graph_database.GraphDatabase method*), 560
intersection_graph() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 692
IntersectionGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 465
IntervalGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 465
IoninKharaghani765Graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 466
is_affine_polar() (*in module sage.graphs.strongly_regular_db*), 580
is_aperiodic() (*sage.graphs.digraph.DiGraph method*), 364
is_apex() (*sage.graphs.graph.Graph method*), 281
is_arc_transitive() (*sage.graphs.graph.Graph method*), 282
is_asteroidal_triple_free() (*in module sage.graphs.asteroidal_triples*), 717
is_asteroidal_triple_free() (*sage.graphs.graph.Graph method*), 282
is_berge_cyclic() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 692
is_biconnected() (*sage.graphs.graph.Graph method*), 283
is_bipartite() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 391
is_bipartite() (*sage.graphs.generic_graph.GenericGraph method*), 105
is_block_graph() (*sage.graphs.graph.Graph method*), 284
is_cactus() (*sage.graphs.graph.Graph method*), 284
is_cartesian_product() (*in module sage.graphs.graph_decompositions.graph_products*), 824
is_cartesian_product() (*sage.graphs.graph.Graph method*), 285
is_cayley() (*sage.graphs.generic_graph.GenericGraph method*), 106
is_chordal() (*sage.graphs.generic_graph.GenericGraph method*), 107
is_circulant() (*sage.graphs.generic_graph.GenericGraph method*), 109
is_circular_planar() (*sage.graphs.generic_graph.GenericGraph method*), 109
is_circumscribable() (*sage.graphs.graph.Graph method*), 286
is_clique() (*sage.graphs.generic_graph.GenericGraph method*), 111

is_cograph() (*sage.graphs.graph.Graph method*), 287
is_comparability() (*in module sage.graphs.comparability*), 723
is_comparability() (*sage.graphs.graph.Graph method*), 287
is_comparability_MILP() (*in module sage.graphs.comparability*), 724
is_complete_multipartite() (*in module sage.graphs.strongly_regular_db*), 580
is_component_connected() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 845
is_connected() (*in module sage.graphs.connectivity*), 944
is_connected() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 692
is_connected() (*sage.graphs.base.c_graph.CGraphBackend method*), 622
is_connected() (*sage.graphs.generic_graph.GenericGraph method*), 112
is_cossidente_penttila() (*in module sage.graphs.strongly_regular_db*), 581
is_cut_edge() (*in module sage.graphs.connectivity*), 944
is_cut_edge() (*sage.graphs.generic_graph.GenericGraph method*), 112
is_cut_vertex() (*in module sage.graphs.connectivity*), 945
is_cut_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 113
is_cycle() (*sage.graphs.generic_graph.GenericGraph method*), 114
is_directed() (*sage.graphs.digraph.DiGraph method*), 365
is_directed() (*sage.graphs.graph.Graph method*), 287
is_directed_acyclic() (*sage.graphs.base.c_graph.CGraphBackend method*), 623
is_directed_acyclic() (*sage.graphs.digraph.DiGraph method*), 365
is_distance_regular() (*in module sage.graphs.distances_all_pairs*), 874
is_distance_regular() (*sage.graphs.graph.Graph method*), 288
is_dominating() (*in module sage.graphs.domination*), 957
is_dominating() (*sage.graphs.graph.Graph method*), 288
is_drawn_free_of_edge_crossings() (*sage.graphs.generic_graph.GenericGraph method*), 115
is_edge_transitive() (*sage.graphs.graph.Graph method*), 289
is_equitable() (*sage.graphs.generic_graph.GenericGraph method*), 115
is_eulerian() (*sage.graphs.generic_graph.GenericGraph method*), 116
is_even_hole_free() (*sage.graphs.graph.Graph method*), 289
is_forest() (*sage.graphs.graph.Graph method*), 290
is_gallai_tree() (*sage.graphs.generic_graph.GenericGraph method*), 117
is_generalized_quadrangle() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 693
is_goethals_seidel() (*in module sage.graphs.strongly_regular_db*), 581
is_GQqmqp() (*in module sage.graphs.strongly_regular_db*), 577
is_haemers() (*in module sage.graphs.strongly_regular_db*), 582
is_half_transitive() (*sage.graphs.graph.Graph method*), 290
is_hamiltonian() (*sage.graphs.generic_graph.GenericGraph method*), 117
is_immutable() (*sage.graphs.generic_graph.GenericGraph method*), 118
is_independent_set() (*sage.graphs.generic_graph.GenericGraph method*), 118
is_inscribable() (*sage.graphs.graph.Graph method*), 291
is_interval() (*sage.graphs.generic_graph.GenericGraph method*), 119
is_isomorphic() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 693
is_isomorphic() (*sage.graphs.generic_graph.GenericGraph method*), 120
is_johnson() (*in module sage.graphs.strongly_regular_db*), 582
is_line_graph() (*in module sage.graphs.line_graph*), 728
is_line_graph() (*sage.graphs.graph.Graph method*), 292
is_long_antihole_free() (*in module sage.graphs.weakly_chordal*), 866
is_long_antihole_free() (*sage.graphs.graph.Graph method*), 293
is_long_hole_free() (*in module sage.graphs.weakly_chordal*), 866

is_long_hole_free() (*sage.graphs.graph.Graph method*), 294
is_mathon_PC_srg() (*in module sage.graphs.strongly_regular_db*), 582
is_muzychuk_S6() (*in module sage.graphs.strongly_regular_db*), 583
is_NO_F2() (*in module sage.graphs.strongly_regular_db*), 578
is_NO_F3() (*in module sage.graphs.strongly_regular_db*), 578
is_NOodd() (*in module sage.graphs.strongly_regular_db*), 578
is_NOperp_F5() (*in module sage.graphs.strongly_regular_db*), 579
is_nowhere0_twoweight() (*in module sage.graphs.strongly_regular_db*), 583
is_NU() (*in module sage.graphs.strongly_regular_db*), 579
is_odd_hole_free() (*sage.graphs.graph.Graph method*), 294
is_orthogonal_array_block_graph() (*in module sage.graphs.strongly_regular_db*), 583
is_orthogonal_polar() (*in module sage.graphs.strongly_regular_db*), 584
is_overfull() (*sage.graphs.graph.Graph method*), 295
is_paley() (*in module sage.graphs.strongly_regular_db*), 585
is_partial_cube() (*in module sage.graphs.partial_cube*), 909
is_partial_cube() (*sage.graphs.graph.Graph method*), 296
is_perfect() (*sage.graphs.graph.Graph method*), 297
is_permutation() (*in module sage.graphs.comparability*), 725
is_permutation() (*sage.graphs.graph.Graph method*), 298
is_planar() (*in module sage.graphs.planarity*), 754
is_planar() (*sage.graphs.generic_graph.GenericGraph method*), 121
is_polhill() (*in module sage.graphs.strongly_regular_db*), 585
is_polyhedral() (*sage.graphs.graph.Graph method*), 299
is_prime() (*sage.graphs.graph.Graph method*), 299
is_redundant() (*in module sage.graphs.domination*), 957
is_redundant() (*sage.graphs.graph.Graph method*), 300
is_regular() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 694
is_regular() (*sage.graphs.generic_graph.GenericGraph method*), 123
is_resolvable() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 694
is_RSHCD() (*in module sage.graphs.strongly_regular_db*), 580
is_self_complementary() (*sage.graphs.generic_graph.GenericGraph method*), 123
is_semi_symmetric() (*sage.graphs.graph.Graph method*), 300
is_simple() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 695
is_split() (*sage.graphs.graph.Graph method*), 301
is_steiner() (*in module sage.graphs.strongly_regular_db*), 586
is_strongly_connected() (*in module sage.graphs.connectivity*), 946
is_strongly_connected() (*sage.graphs.base.c_graph.CGraphBackend method*), 624
is_strongly_connected() (*sage.graphs.digraph.DiGraph method*), 366
is_strongly_regular() (*in module sage.graphs.base.static_dense_graph*), 650
is_strongly_regular() (*sage.graphs.graph.Graph method*), 301
is_subgraph() (*sage.graphs.generic_graph.GenericGraph method*), 124
is_switch_OA_srg() (*in module sage.graphs.strongly_regular_db*), 586
is_switch_skewhad() (*in module sage.graphs.strongly_regular_db*), 586
is_t_design() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 695
is_taylor_twograph_srg() (*in module sage.graphs.strongly_regular_db*), 587
is_tournament() (*sage.graphs.digraph.DiGraph method*), 366
is_transitive() (*in module sage.graphs.comparability*), 725
is_transitive() (*sage.graphs.digraph.DiGraph method*), 367
is_transitively_reduced() (*sage.graphs.generic_graph.GenericGraph method*), 125
is_tree() (*sage.graphs.graph.Graph method*), 303

is_triangle_free() (*in module sage.graphs.base.static_dense_graph*), 652
is_triangle_free() (*sage.graphs.graph.Graph method*), 304
is_triconnected() (*in module sage.graphs.connectivity*), 946
is_triconnected() (*sage.graphs.graph.Graph method*), 304
is_twograph_descendant_of_srg() (*in module sage.graphs.strongly_regular_db*), 587
is_uniform() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 697
is_unitary_dual_polar() (*in module sage.graphs.strongly_regular_db*), 588
is_unitary_polar() (*in module sage.graphs.strongly_regular_db*), 588
is_valid_lex_M_order() (*in module sage.graphs.traversals*), 755
is_valid_ordering() (*in module sage.graphs.graph_decompositions.vertex_separation*), 803
is_vertex_transitive() (*sage.graphs.generic_graph.GenericGraph method*), 125
is_weakly_chordal() (*in module sage.graphs.weakly_chordal*), 867
is_weakly_chordal() (*sage.graphs.graph.Graph method*), 305
isomorphic_substructures_iterator() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 697
iterator_edges() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 647
iterator_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
iterator_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 639
iterator_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_in_edges() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 647
iterator_in_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 667
iterator_in_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 640
iterator_in_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_in_nbrs() (*sage.graphs.base.c_graph.CGraphBackend method*), 624
iterator_in_nbrs() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 668
iterator_in_nbrs() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_nbrs() (*sage.graphs.base.c_graph.CGraphBackend method*), 625
iterator_nbrs() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 668
iterator_nbrs() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_out_edges() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 648
iterator_out_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 668
iterator_out_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 640
iterator_out_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_out_nbrs() (*sage.graphs.base.c_graph.CGraphBackend method*), 625
iterator_out_nbrs() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 668
iterator_out_nbrs() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 662
iterator_unsorted_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 640
iterator_verts() (*sage.graphs.base.c_graph.CGraphBackend method*), 625
iterator_verts() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
iterator_verts() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663

J

JankoKharaghaniGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 468
JankoKharaghaniTonchevGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 468
johnson_closeness_centrality() (*in module sage.graphs.base.boost_graph*), 675
johnson_shortest_paths() (*in module sage.graphs.base.boost_graph*), 676
JohnsonGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 468
join() (*sage.graphs.graph.Graph method*), 306

K

katz_centrality() (*sage.graphs.generic_graph.GenericGraph method*), 126
katz_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 126
Kautz() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 548
KingGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 469
kirchhoff_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 128
kirchhoff_symanzik_polynomial() (*sage.graphs.graph.Graph method*), 307
KittellGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 469
Klein3RegularGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 470
Klein7RegularGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 470
KneserGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 470
KnightGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 471
KrackhardtKiteGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 471
kronecker_product() (*sage.graphs.generic_graph.GenericGraph method*), 130
kruskal() (*in module sage.graphs.spanning_tree*), 734
kruskal_iterator() (*in module sage.graphs.spanning_tree*), 736
kruskal_iterator_from_edges() (*in module sage.graphs.spanning_tree*), 736

L

LadderGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 473
laplacian_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 130
latex() (*sage.graphs.graph_latex.GraphLatex method*), 883
latex_options() (*sage.graphs.generic_graph.GenericGraph method*), 132
latin_squares_graph_parameters() (*in module sage.graphs.strongly_regular_db*), 588
layout() (*sage.graphs.generic_graph.GenericGraph method*), 132
layout_acyclic() (*sage.graphs.digraph.DiGraph method*), 367
layout_acyclic_dummy() (*sage.graphs.digraph.DiGraph method*), 368
layout_circular() (*sage.graphs.generic_graph.GenericGraph method*), 134
layout_default() (*sage.graphs.generic_graph.GenericGraph method*), 135
layout_extend_randomly() (*sage.graphs.generic_graph.GenericGraph method*), 135
layout_graphviz() (*sage.graphs.generic_graph.GenericGraph method*), 136
layout_planar() (*sage.graphs.generic_graph.GenericGraph method*), 137
layout_ranked() (*sage.graphs.generic_graph.GenericGraph method*), 138
layout_split() (*in module sage.graphs.generic_graph_pyx*), 926
layout_spring() (*sage.graphs.generic_graph.GenericGraph method*), 139
layout_tree() (*sage.graphs.generic_graph.GenericGraph method*), 139
layout_tree() (*sage.graphs.graph_plot.GraphPlot method*), 772
LCFGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 472
least_effective_resistance() (*sage.graphs.graph.Graph method*), 308
length_and_string_from_graph6() (*in module sage.graphs.generic_graph_pyx*), 927
level_sets() (*sage.graphs.digraph.DiGraph method*), 368
lex_BFS() (*in module sage.graphs.traversals*), 755
lex_BFS() (*sage.graphs.generic_graph.GenericGraph method*), 140
lex_DFS() (*in module sage.graphs.traversals*), 757
lex_DFS() (*sage.graphs.generic_graph.GenericGraph method*), 142
lex_DOWN() (*in module sage.graphs.traversals*), 758
lex_DOWN() (*sage.graphs.generic_graph.GenericGraph method*), 143
lex_M() (*in module sage.graphs.traversals*), 759
lex_M() (*sage.graphs.graph.Graph method*), 308
lex_M_fast() (*in module sage.graphs.traversals*), 760

lex_M_slow() (*in module sage.graphs.traversals*), 761
lex_UP() (*in module sage.graphs.traversals*), 762
lex_UP() (*sage.graphs.generic_graph.GenericGraph method*), 144
lexicographic_product() (*sage.graphs.generic_graph.GenericGraph method*), 145
line_graph() (*in module sage.graphs.line_graph*), 729
line_graph() (*sage.graphs.generic_graph.GenericGraph method*), 145
line_graph_forbidden_subgraphs() (*sage.graphs.graph_generators.GraphGenerators static method*), 531
linear_arboricity() (*in module sage.graphs.graph_coloring*), 708
linear_ordering_to_path_decomposition() (*in module sage.graphs.graph_decompositions.vertex_separation*), 803
LivingstoneGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 473
LjubljanaGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 473
load_afile() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 391
LocalMcLaughlinGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 474
LollipopGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 474
longest_path() (*sage.graphs.generic_graph.GenericGraph method*), 146
loop_edges() (*sage.graphs.generic_graph.GenericGraph method*), 149
loop_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 150
loops() (*sage.graphs.base.c_graph.CGraphBackend method*), 626
loops() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
loops() (*sage.graphs.generic_graph.GenericGraph method*), 150
lovasz_theta() (*in module sage.graphs.lovasz_theta*), 751
lovasz_theta() (*sage.graphs.graph.Graph method*), 310
lower_bound() (*in module sage.graphs.graph_decompositions.vertex_separation*), 804

M

M22Graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 474
magnitude_function() (*sage.graphs.graph.Graph method*), 310
make_labelled_rooted_tree() (*in module sage.graphs.graph_decompositions.clique_separators*), 862
make_tree() (*in module sage.graphs.graph_decompositions.clique_separators*), 862
MarkstroemGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 475
matching() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 392
matching() (*sage.graphs.graph.Graph method*), 311
matching_polynomial() (*in module sage.graphs.matchpoly*), 745
matching_polynomial() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 392
matching_polynomial() (*sage.graphs.graph.Graph method*), 312
MathonPseudocyclicMergingGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 475
MathonPseudocyclicStronglyRegularGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 475
MathonStronglyRegularGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 477
max_clique() (*in module sage.graphs.cliquer*), 714
max_cut() (*sage.graphs.generic_graph.GenericGraph method*), 151
maximal_subtrees_with_leaves_in_x() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 845
MaximizeDegree (*class in sage.graphs.tutte_polynomial*), 904
maximum_average_degree() (*sage.graphs.graph.Graph method*), 316
maximum_cardinality_search() (*in module sage.graphs.traversals*), 763
maximum_cardinality_search() (*sage.graphs.graph.Graph method*), 316

maximum_cardinality_search_M() (*in module sage.graphs.traversals*), 764
maximum_cardinality_search_M() (*sage.graphs.graph.Graph method*), 317
McGeeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 477
McLaughlinGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 477
md_tree_to_graph() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 847
MeredithGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 478
merge_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 152
min_cycle_basis() (*in module sage.graphs.base.boost_graph*), 677
min_spanning_tree() (*in module sage.graphs.base.boost_graph*), 678
min_spanning_tree() (*sage.graphs.generic_graph.GenericGraph method*), 153
minimal_dominating_sets() (*in module sage.graphs.domination*), 957
minimal_dominating_sets() (*sage.graphs.graph.Graph method*), 319
minimal_schnyder_wood() (*in module sage.graphs.schnyder*), 753
MinimizeDegree (*class in sage.graphs.tutte_polynomial*), 904
MinimizeSingleDegree (*class in sage.graphs.tutte_polynomial*), 904
minimum_cycle_basis() (*sage.graphs.generic_graph.GenericGraph method*), 155
minimum_outdegree_orientation() (*sage.graphs.graph.Graph method*), 320
minor() (*sage.graphs.graph.Graph method*), 321
mkgraph() (*in module sage.graphs.graph_decompositions.rankwidth*), 814
modular_decomposition() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 847
modular_decomposition() (*sage.graphs.graph.Graph method*), 322
MoebiusKantorGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 478
MoserSpindle() (*sage.graphs.graph_generators.GraphGenerators static method*), 478
most_common_neighbors() (*sage.graphs.graph.Graph method*), 324
multicommodity_flow() (*sage.graphs.generic_graph.GenericGraph method*), 156
multiple_edges() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 648
multiple_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
multiple_edges() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 641
multiple_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
multiple_edges() (*sage.graphs.generic_graph.GenericGraph method*), 157
multiway_cut() (*sage.graphs.generic_graph.GenericGraph method*), 158
MuzychukS6Graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 479
MycielskiGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 480
MycielskiStep() (*sage.graphs.graph_generators.GraphGenerators static method*), 481

N

name() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
name() (*sage.graphs.generic_graph.GenericGraph method*), 159
NauruGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 482
nauty() (*sage.graphs.hypergraph_generators.HypergraphGenerators method*), 682
nauty_directg() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 553
nauty_geng() (*sage.graphs.graph_generators.GraphGenerators method*), 531
neighbor_in_iterator() (*sage.graphs.digraph.DiGraph method*), 369
neighbor_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 159
neighbor_out_iterator() (*sage.graphs.digraph.DiGraph method*), 369
neighbors() (*sage.graphs.generic_graph.GenericGraph method*), 160
neighbors_in() (*sage.graphs.digraph.DiGraph method*), 370
neighbors_out() (*sage.graphs.digraph.DiGraph method*), 370
nested_tuple_to_tree() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 848
networkx_graph() (*sage.graphs.generic_graph.GenericGraph method*), 160

new_P() (*in module sage.graphs.pq_trees*), 743
new_Q() (*in module sage.graphs.pq_trees*), 743
NKStarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 481
Node (*class in sage.graphs.graph_decompositions.modular_decomposition*), 825
NodeSplit (*class in sage.graphs.graph_decompositions.modular_decomposition*), 826
NodeType (*class in sage.graphs.graph_decompositions.modular_decomposition*), 827
NonisotropicOrthogonalPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 482
NonisotropicUnitaryPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 483
Nowhere0WordsTwoWeightCodeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 484
nowhere_zero_flow() (*sage.graphs.generic_graph.GenericGraph method*), 161
NStarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 481
num_blocks() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 698
num_edges() (*sage.graphs.base.c_graph.CGraphBackend method*), 626
num_edges() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
num_edges() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
num_edges() (*sage.graphs.generic_graph.GenericGraph method*), 162
num_faces() (*sage.graphs.generic_graph.GenericGraph method*), 162
num_points() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 699
num_verts() (*sage.graphs.base.c_graph.CGraphBackend method*), 627
num_verts() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
num_verts() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
num_verts() (*sage.graphs.generic_graph.GenericGraph method*), 162
number_cocomponents() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 849
number_components() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 849
number_of() (*sage.graphs.graph_database.GraphQuery method*), 565
number_of_children() (*sage.graphs.pq_trees.PQ method*), 741
number_of_loops() (*sage.graphs.generic_graph.GenericGraph method*), 163
number_of_n_colorings() (*in module sage.graphs.graph_coloring*), 710
numbers_of_colorings() (*in module sage.graphs.graph_coloring*), 710

O

OctahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 484
odd_girth() (*sage.graphs.generic_graph.GenericGraph method*), 163
OddGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 485
order() (*sage.graphs.generic_graph.GenericGraph method*), 164
ordering() (*sage.graphs.pq_trees.PQ method*), 741
orderings() (*sage.graphs.pq_trees.P method*), 739
orderings() (*sage.graphs.pq_trees.Q method*), 742
orientations() (*sage.graphs.graph.Graph method*), 325
OrthogonalArrayBlockGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 485
OrthogonalPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 487
out_branchings() (*sage.graphs.digraph.DiGraph method*), 370
out_degree() (*sage.graphs.base.c_graph.CGraphBackend method*), 627
out_degree() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
out_degree() (*sage.graphs.base.sparse_graph.SparseGraph method*), 636
out_degree() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
out_degree() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664
out_degree() (*sage.graphs.digraph.DiGraph method*), 371

out_degree_iterator() (*sage.graphs.digraph.DiGraph method*), 372
out_degree_sequence() (*sage.graphs.digraph.DiGraph method*), 372
out_neighbors() (*sage.graphs.base.c_graph.CGraph method*), 612
out_neighbors() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664
outgoing_edge_iterator() (*sage.graphs.digraph.DiGraph method*), 372
outgoing_edges() (*sage.graphs.digraph.DiGraph method*), 373

P

P (*class in sage.graphs.pq_trees*), 739
packing() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 699
pagerank() (*sage.graphs.generic_graph.GenericGraph method*), 164
Paley() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 549
PaleyGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 487
PappusGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 488
PasechnikGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 488
Path() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 549
path_decomposition() (*in module sage.graphs.graph_decompositions.vertex_separation*), 804
path_semigroup() (*sage.graphs.digraph.DiGraph method*), 373
PathGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 488
pathwidth() (*in module sage.graphs.graph_decompositions.vertex_separation*), 806
pathwidth() (*sage.graphs.graph.Graph method*), 325
perfect_matchings() (*sage.graphs.graph.Graph method*), 326
period() (*sage.graphs.digraph.DiGraph method*), 373
periphery() (*sage.graphs.generic_graph.GenericGraph method*), 167
PerkelGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 489
PermutationGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 489
permute_decomposition() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 850
petersen_family() (*sage.graphs.graph_generators.GraphGenerators static method*), 533
PetersenGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 491
planar_dual() (*sage.graphs.generic_graph.GenericGraph method*), 167
planar_graphs() (*sage.graphs.graph_generators.GraphGenerators method*), 533
plot() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 393
plot() (*sage.graphs.generic_graph.GenericGraph method*), 168
plot() (*sage.graphs.graph_plot.GraphPlot method*), 772
plot3d() (*sage.graphs.generic_graph.GenericGraph method*), 172
PoussinGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 491
PQ (*class in sage.graphs.pq_trees*), 740
print_md_tree() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 850
print_triconnected_components() (*sage.graphs.connectivity.TriconnectivitySPQR method*), 935
private_neighbors() (*in module sage.graphs.domination*), 959
private_neighbors() (*sage.graphs.graph.Graph method*), 327
project_left() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 393
project_right() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 393
promote_child() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 851
promote_left() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 851
promote_right() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 852

Q

Q (*class in sage.graphs.pq_trees*), 742
quadrangulations() (*sage.graphs.graph_generators.GraphGenerators method*), 534

QueenGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 492
query() (*sage.graphs.graph_database.GraphDatabase method*), 561
query_iterator() (*sage.graphs.graph_database.GraphQuery method*), 565

R

radius() (*sage.graphs.generic_graph.GenericGraph method*), 174
random() (*sage.graphs.graph_coloring.Test method*), 701
random_all_graph_colorings() (*sage.graphs.graph_coloring.Test method*), 702
random_edge() (*sage.graphs.generic_graph.GenericGraph method*), 175
random_edge_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 175
random_md_tree() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 853
random_orientation() (*in module sage.graphs.orientations*), 929
random_orientation() (*sage.graphs.graph.Graph method*), 328
random_spanning_tree() (*in module sage.graphs.spanning_tree*), 737
random_spanning_tree() (*sage.graphs.graph.Graph method*), 328
random_subgraph() (*sage.graphs.generic_graph.GenericGraph method*), 176
random_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 176
random_vertex_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 176
RandomBarabasiAlbert() (*sage.graphs.graph_generators.GraphGenerators static method*), 493
RandomBicubicPlanar() (*sage.graphs.graph_generators.GraphGenerators static method*), 493
RandomBipartite() (*sage.graphs.graph_generators.GraphGenerators static method*), 494
RandomBlockGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 495
RandomBoundedToleranceGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 496
RandomChordalGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 496
RandomDirectedGN() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 550
RandomDirectedGNC() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 550
RandomDirectedGNM() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 550
RandomDirectedGNP() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 551
RandomDirectedGNR() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 551
RandomGNM() (*sage.graphs.graph_generators.GraphGenerators static method*), 498
RandomGNP() (*in module sage.graphs.graph_generators_pyx*), 555
RandomGNP() (*sage.graphs.graph_generators.GraphGenerators static method*), 498
RandomHolmeKim() (*sage.graphs.graph_generators.GraphGenerators static method*), 499
RandomIntervalGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 500
RandomLobster() (*sage.graphs.graph_generators.GraphGenerators static method*), 500
RandomNewmanWattsStrogatz() (*sage.graphs.graph_generators.GraphGenerators static method*), 501
RandomRegular() (*sage.graphs.graph_generators.GraphGenerators static method*), 501
RandomRegularBipartite() (*sage.graphs.graph_generators.GraphGenerators static method*), 502
RandomSemiComplete() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 552
RandomShell() (*sage.graphs.graph_generators.GraphGenerators static method*), 502
RandomToleranceGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 503
RandomTournament() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 552
RandomTree() (*sage.graphs.graph_generators.GraphGenerators static method*), 503
RandomTreePowerlaw() (*sage.graphs.graph_generators.GraphGenerators static method*), 503
RandomTriangulation() (*sage.graphs.graph_generators.GraphGenerators static method*), 504
rank() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 699
rank_decomposition() (*in module sage.graphs.graph_decompositions.rankwidth*), 814
rank_decomposition() (*sage.graphs.graph.Graph method*), 329
realloc() (*sage.graphs.base.c_graph.CGraph method*), 613
realloc() (*sage.graphs.base.dense_graph.DenseGraph method*), 644

realloc() (*sage.graphs.base.sparse_graph.SparseGraph method*), 636
recreate_decomposition() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 853
recursively_number_parts() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 854
reduced_adjacency_matrix() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 394
refine() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 854
relabel() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 699
relabel() (*sage.graphs.base.c_graph.CGraphBackend method*), 627
relabel() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
relabel() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
relabel() (*sage.graphs.generic_graph.GenericGraph method*), 177
relabel_tree() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 855
remove_loops() (*sage.graphs.generic_graph.GenericGraph method*), 179
remove_multiple_edges() (*sage.graphs.generic_graph.GenericGraph method*), 180
removed_edge() (*in module sage.graphs.tutte_polynomial*), 905
removed_from() (*sage.graphs.tutte_polynomial.Ear method*), 903
removed_loops() (*in module sage.graphs.tutte_polynomial*), 905
removed_multiedge() (*in module sage.graphs.tutte_polynomial*), 906
reorder_sets() (*in module sage.graphs.pq_trees*), 743
reverse() (*sage.graphs.digraph.DiGraph method*), 374
reverse() (*sage.graphs.pq_trees.PQ method*), 741
reverse_edge() (*sage.graphs.digraph.DiGraph method*), 374
reverse_edges() (*sage.graphs.digraph.DiGraph method*), 376
RingedTree() (*sage.graphs.graph_generators.GraphGenerators static method*), 505
RobertsonGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 505
RookGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 506
root_graph() (*in module sage.graphs.line_graph*), 730
round_robin() (*in module sage.graphs.graph_coloring*), 710

S

s (*sage.graphs.tutte_polynomial.Ear attribute*), 903
sage.combinat.designs.incidence_structures (*module*), 683
sage.graphs.asteroidal_triples (*module*), 716
sage.graphs.base.boost_graph (*module*), 670
sage.graphs.base.c_graph (*module*), 602
sage.graphs.base.dense_graph (*module*), 641
sage.graphs.base.graph_backends (*module*), 664
sage.graphs.base.overview (*module*), 601
sage.graphs.base.sparse_graph (*module*), 630
sage.graphs.base.static_dense_graph (*module*), 649
sage.graphs.base.static_sparse_backend (*module*), 658
sage.graphs.base.static_sparse_graph (*module*), 652
sage.graphs.bipartite_graph (*module*), 383
sage.graphs.centrality (*module*), 714
sage.graphs.cliquer (*module*), 712
sage.graphs.comparability (*module*), 720
sage.graphs.connectivity (*module*), 930
sage.graphs.convexity_properties (*module*), 862
sage.graphs.digraph (*module*), 342
sage.graphs.digraph_generators (*module*), 542
sage.graphs.distances_all_pairs (*module*), 868

sage.graphs.domination (*module*), 955
sage.graphs.generic_graph (*module*), 1
sage.graphs.generic_graph_pyx (*module*), 923
sage.graphs.genus (*module*), 748
sage.graphs.graph (*module*), 228
sage.graphs.graph_coloring (*module*), 701
sage.graphs.graph_database (*module*), 555
sage.graphs.graph_decompositions.bandwidth (*module*), 815
sage.graphs.graph_decompositions.clique_separators (*module*), 858
sage.graphs.graph_decompositions.cutwidth (*module*), 817
sage.graphs.graph_decompositions.graph_products (*module*), 822
sage.graphs.graph_decompositions.modular_decomposition (*module*), 825
sage.graphs.graph_decompositions.rankwidth (*module*), 812
sage.graphs.graph_decompositions.vertex_separation (*module*), 800
sage.graphs.graph_editor (*module*), 890
sage.graphs.graph_generators (*module*), 403
sage.graphs.graph_generators_pyx (*module*), 555
sage.graphs.graph_input (*module*), 894
sage.graphs.graph_latex (*module*), 876
sage.graphs.graph_list (*module*), 890
sage.graphs.graph_plot (*module*), 766
sage.graphs.graph_plot_js (*module*), 797
sage.graphs.hyperbolicity (*module*), 897
sage.graphs.hypergraph_generators (*module*), 681
sage.graphs.independent_sets (*module*), 718
sage.graphs.isgci (*module*), 591
sage.graphs.line_graph (*module*), 726
sage.graphs.lovasz_theta (*module*), 751
sage.graphs.matchpoly (*module*), 745
sage.graphs.orientations (*module*), 929
sage.graphs.partial_cube (*module*), 907
sage.graphs.path_enumeration (*module*), 909
sage.graphs.planarity (*module*), 754
sage.graphs.pq_trees (*module*), 738
sage.graphs.schnyder (*module*), 751
sage.graphs.spanning_tree (*module*), 730
sage.graphs.strongly_regular_db (*module*), 569
sage.graphs.traversals (*module*), 755
sage.graphs.trees (*module*), 744
sage.graphs.tutte_polynomial (*module*), 902
sage.graphs.views (*module*), 396
sage.graphs.weakly_chordal (*module*), 865
save_afile() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 394
SchlaefliGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 506
Search_iterator (*class in sage.graphs.base.c_graph*), 630
seidel_adjacency_matrix() (*sage.graphs.graph.Graph method*), 329
seidel_switching() (*sage.graphs.graph.Graph method*), 330
set_contiguous() (*in module sage.graphs.pq_trees*), 744
set_contiguous() (*sage.graphs.pq_trees.P method*), 740
set_contiguous() (*sage.graphs.pq_trees.Q method*), 742

set_edge_label() (*sage.graphs.base.dense_graph.DenseGraphBackend method*), 648
set_edge_label() (*sage.graphs.base.graph_backends.GenericGraphBackend method*), 669
set_edge_label() (*sage.graphs.base.sparse_graph.SparseGraphBackend method*), 641
set_edge_label() (*sage.graphs.base.static_sparse_backend.StaticSparseBackend method*), 663
set_edge_label() (*sage.graphs.generic_graph.GenericGraph method*), 180
set_edges() (*sage.graphs.graph_plot.GraphPlot method*), 789
set_embedding() (*sage.graphs.generic_graph.GenericGraph method*), 182
set_latex_options() (*sage.graphs.generic_graph.GenericGraph method*), 182
set_node_split() (*sage.graphs.graph_decompositions.modular_decomposition.Node method*), 826
set_option() (*sage.graphs.graph_latex.GraphLatex method*), 884
set_options() (*sage.graphs.graph_latex.GraphLatex method*), 888
set_planar_positions() (*sage.graphs.generic_graph.GenericGraph method*), 182
set_pos() (*sage.graphs.generic_graph.GenericGraph method*), 183
set_pos() (*sage.graphs.graph_plot.GraphPlot method*), 790
set_vertex() (*sage.graphs.generic_graph.GenericGraph method*), 183
set_vertices() (*sage.graphs.generic_graph.GenericGraph method*), 183
set_vertices() (*sage.graphs.graph_plot.GraphPlot method*), 794
setup_latex_preamble() (*in module sage.graphs.graph_latex*), 889
shortest_path() (*sage.graphs.base.c_graph.CGraphBackend method*), 628
shortest_path() (*sage.graphs.generic_graph.GenericGraph method*), 184
shortest_path_all_pairs() (*in module sage.graphs.distances_all_pairs*), 875
shortest_path_all_pairs() (*sage.graphs.generic_graph.GenericGraph method*), 185
shortest_path_all_vertices() (*sage.graphs.base.c_graph.CGraphBackend method*), 628
shortest_path_length() (*sage.graphs.generic_graph.GenericGraph method*), 189
shortest_path_lengths() (*sage.graphs.generic_graph.GenericGraph method*), 190
shortest_path_special() (*sage.graphs.base.c_graph.CGraphBackend method*), 629
shortest_paths() (*in module sage.graphs.base.boost_graph*), 678
shortest_paths() (*sage.graphs.generic_graph.GenericGraph method*), 192
shortest_simple_paths() (*in module sage.graphs.path_enumeration*), 919
shortest_simple_paths() (*sage.graphs.generic_graph.GenericGraph method*), 194
show() (*sage.graphs.generic_graph.GenericGraph method*), 195
show() (*sage.graphs.graph_database.GraphQuery method*), 565
show() (*sage.graphs.graph_plot.GraphPlot method*), 795
show3d() (*sage.graphs.generic_graph.GenericGraph method*), 196
show_all() (*sage.graphs.isgci.GraphClasses method*), 599
show_graphs() (*in module sage.graphs.graph_list*), 891
ShrikhandeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 507
SierpinskiGasketGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 508
simple_connected_genus_backtracker (*class in sage.graphs.genus*), 748
simple_connected_graph_genus() (*in module sage.graphs.genus*), 750
simplify() (*sage.graphs.pq_trees.PQ method*), 741
SimsGewirtzGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 510
sinks() (*sage.graphs.digraph.DiGraph method*), 378
size() (*sage.graphs.generic_graph.GenericGraph method*), 197
small_integer_to_graph6() (*in module sage.graphs.generic_graph_pyx*), 927
smallgraphs() (*sage.graphs.isgci.GraphClasses method*), 599
sources() (*sage.graphs.digraph.DiGraph method*), 378
SousselierGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 510
spanning_trees() (*sage.graphs.graph.Graph method*), 330
spanning_trees_count() (*sage.graphs.generic_graph.GenericGraph method*), 197

sparse6_string() (*sage.graphs.graph.Graph method*), 331
SparseGraph (*class in sage.graphs.base.sparse_graph*), 633
SparseGraphBackend (*class in sage.graphs.base.sparse_graph*), 637
spectral_radius() (*in module sage.graphs.base.static_sparse_graph*), 655
spectral_radius() (*sage.graphs.generic_graph.GenericGraph method*), 198
spectrum() (*sage.graphs.generic_graph.GenericGraph method*), 200
spqr_tree() (*in module sage.graphs.connectivity*), 947
spqr_tree() (*sage.graphs.graph.Graph method*), 331
spqr_tree_to_graph() (*in module sage.graphs.connectivity*), 950
spring_layout_fast() (*in module sage.graphs.generic_graph_pyx*), 927
spring_layout_fast_split() (*in module sage.graphs.generic_graph_pyx*), 928
SquaredSkewHadamardMatrixGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 511
SRG_100_44_18_20() (*in module sage.graphs.strongly_regular_db*), 569
SRG_100_45_20_20() (*in module sage.graphs.strongly_regular_db*), 569
SRG_105_32_4_12() (*in module sage.graphs.strongly_regular_db*), 569
SRG_120_63_30_36() (*in module sage.graphs.strongly_regular_db*), 570
SRG_120_77_52_44() (*in module sage.graphs.strongly_regular_db*), 570
SRG_126_25_8_4() (*in module sage.graphs.strongly_regular_db*), 570
SRG_126_50_13_24() (*in module sage.graphs.strongly_regular_db*), 570
SRG_1288_792_476_504() (*in module sage.graphs.strongly_regular_db*), 571
SRG_144_39_6_12() (*in module sage.graphs.strongly_regular_db*), 571
SRG_175_72_20_36() (*in module sage.graphs.strongly_regular_db*), 571
SRG_176_105_68_54() (*in module sage.graphs.strongly_regular_db*), 571
SRG_176_49_12_14() (*in module sage.graphs.strongly_regular_db*), 572
SRG_176_90_38_54() (*in module sage.graphs.strongly_regular_db*), 572
SRG_196_91_42_42() (*in module sage.graphs.strongly_regular_db*), 572
SRG_210_99_48_45() (*in module sage.graphs.strongly_regular_db*), 572
SRG_220_84_38_28() (*in module sage.graphs.strongly_regular_db*), 573
SRG_243_110_37_60() (*in module sage.graphs.strongly_regular_db*), 573
SRG_253_140_87_65() (*in module sage.graphs.strongly_regular_db*), 573
SRG_276_140_58_84() (*in module sage.graphs.strongly_regular_db*), 573
SRG_280_117_44_52() (*in module sage.graphs.strongly_regular_db*), 573
SRG_280_135_70_60() (*in module sage.graphs.strongly_regular_db*), 574
SRG_416_100_36_20() (*in module sage.graphs.strongly_regular_db*), 574
SRG_560_208_72_80() (*in module sage.graphs.strongly_regular_db*), 574
SRG_630_85_20_10() (*in module sage.graphs.strongly_regular_db*), 574
SRG_from_RSHCD() (*in module sage.graphs.strongly_regular_db*), 575
StarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 511
StaticSparseBackend (*class in sage.graphs.base.static_sparse_backend*), 658
StaticSparseCGraph (*class in sage.graphs.base.static_sparse_backend*), 663
steiner_tree() (*sage.graphs.generic_graph.GenericGraph method*), 200
strong_articulation_points() (*in module sage.graphs.connectivity*), 950
strong_articulation_points() (*sage.graphs.digraph.DiGraph method*), 378
strong_orientation() (*sage.graphs.graph.Graph method*), 333
strong_orientations_iterator() (*in module sage.graphs.orientations*), 929
strong_orientations_iterator() (*sage.graphs.graph.Graph method*), 334
strong_product() (*sage.graphs.generic_graph.GenericGraph method*), 202
strongly_connected_component_containing_vertex() (*in module sage.graphs.connectivity*), 951
strongly_connected_component_containing_vertex() (*sage.graphs.base.c_graph.CGraphBackend*

method), 629
strongly_connected_component_containing_vertex() (*sage.graphs.digraph.DiGraph method*), 379
strongly_connected_components() (*sage.graphs.digraph.DiGraph method*), 379
strongly_connected_components_digraph() (*in module sage.graphs.base.static_sparse_graph*), 657
strongly_connected_components_digraph() (*in module sage.graphs.connectivity*), 951
strongly_connected_components_digraph() (*sage.graphs.digraph.DiGraph method*), 380
strongly_connected_components_subgraphs() (*in module sage.graphs.connectivity*), 953
strongly_connected_components_subgraphs() (*sage.graphs.digraph.DiGraph method*), 381
strongly_regular_from_two_intersection_set() (*in module sage.graphs.strongly_regular_db*), 589
strongly_regular_from_two_weight_code() (*in module sage.graphs.strongly_regular_db*), 589
strongly_regular_graph() (*in module sage.graphs.strongly_regular_db*), 590
strongly_regular_graph() (*sage.graphs.graph_generators.GraphGenerators static method*), 536
strongly_regular_graph_lazy() (*in module sage.graphs.strongly_regular_db*), 591
subdivide_edge() (*sage.graphs.generic_graph.GenericGraph method*), 202
subdivide_edges() (*sage.graphs.generic_graph.GenericGraph method*), 203
subgraph() (*sage.graphs.generic_graph.GenericGraph method*), 204
subgraph_search() (*sage.graphs.generic_graph.GenericGraph method*), 206
subgraph_search_count() (*sage.graphs.generic_graph.GenericGraph method*), 208
subgraph_search_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 209
subgraphs_to_query() (*in module sage.graphs.graph_database*), 568
SubgraphSearch (*class in sage.graphs.generic_graph_pyx*), 923
SuzukiGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 512
SwitchedSquaredSkewHadamardMatrixGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 512
SylvesterGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 513
SymplecticDualPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 513
SymplecticPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 513
szeged_index() (*sage.graphs.generic_graph.GenericGraph method*), 210
SzekeresSnarkGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 514

T

T2starGeneralizedQuadrangleGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 514
tachyon_vertex_plot() (*in module sage.graphs.generic_graph*), 227
TadpoleGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 515
tarjan_strongly_connected_components() (*in module sage.graphs.base.static_sparse_graph*), 657
TaylorTwographDescendantSRG() (*sage.graphs.graph_generators.GraphGenerators static method*), 515
TaylorTwographSRG() (*sage.graphs.graph_generators.GraphGenerators static method*), 516
tensor_product() (*sage.graphs.generic_graph.GenericGraph method*), 210
Test (*class in sage.graphs.graph_coloring*), 701
test_gamma_modules() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 855
test_maximal_modules() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 856
test_modular_decomposition() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 856
test_module() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 856
TetrahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 516
ThomsenGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 517
TietzeGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 517
tkz_picture() (*sage.graphs.graph_latex.GraphLatex method*), 888
to_dictionary() (*sage.graphs.generic_graph.GenericGraph method*), 211

to_directed() (*sage.graphs.digraph.DiGraph method*), 381
to_directed() (*sage.graphs.graph.Graph method*), 335
to_graph6() (*in module sage.graphs.graph_list*), 892
to_graphics_array() (*in module sage.graphs.graph_list*), 893
to_simple() (*sage.graphs.generic_graph.GenericGraph method*), 212
to_sparse6() (*in module sage.graphs.graph_list*), 893
to_undirected() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 395
to_undirected() (*sage.graphs.digraph.DiGraph method*), 381
to_undirected() (*sage.graphs.graph.Graph method*), 335
ToleranceGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 517
topological_minor() (*sage.graphs.graph.Graph method*), 336
topological_sort() (*sage.graphs.digraph.DiGraph method*), 382
topological_sort_generator() (*sage.graphs.digraph.DiGraph method*), 383
Toroidal6RegularGrid2dGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 518
ToroidalGrid2dGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 519
tournaments_nauty() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 554
trace() (*sage.combinat.designs.incidence_structures.IncidenceStructure method*), 700
transitive_closure() (*sage.graphs.generic_graph.GenericGraph method*), 213
transitive_reduction() (*sage.graphs.generic_graph.GenericGraph method*), 214
transitive_reduction_acyclic() (*in module sage.graphs.generic_graph_pyx*), 928
TransitiveTournament() (*sage.graphs.digraph_generators.DiGraphGenerators method*), 553
traveling_salesman_problem() (*sage.graphs.generic_graph.GenericGraph method*), 214
tree_to_nested_tuple() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 857
TreeIterator (*class in sage.graphs.trees*), 744
TreeNode (*class in sage.graphs.schnyder*), 752
trees() (*sage.graphs.graph_generators.GraphGenerators static method*), 537
treewidth() (*sage.graphs.graph.Graph method*), 337
triangles_count() (*in module sage.graphs.base.static_dense_graph*), 652
triangles_count() (*in module sage.graphs.base.static_sparse_graph*), 658
triangles_count() (*sage.graphs.generic_graph.GenericGraph method*), 216
triangulations() (*sage.graphs.graph_generators.GraphGenerators method*), 538
TriconnectivitySPQR (*class in sage.graphs.connectivity*), 931
TruncatedIcosidodecahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 519
TruncatedTetrahedralGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 519
TuranGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 520
Tutte12Cage() (*sage.graphs.graph_generators.GraphGenerators static method*), 520
tutte_polynomial() (*in module sage.graphs.tutte_polynomial*), 906
tutte_polynomial() (*sage.graphs.graph.Graph method*), 337
TutteCoxeterGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 520
TutteGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 521
two_factor_petersen() (*sage.graphs.graph.Graph method*), 338
twograph() (*sage.graphs.graph.Graph method*), 339

U

U42Graph216() (*sage.graphs.graph_generators.GraphGenerators static method*), 521
U42Graph540() (*sage.graphs.graph_generators.GraphGenerators static method*), 521
underlying_graph() (*in module sage.graphs.tutte_polynomial*), 907
UniformRandomUniform() (*sage.graphs.hypergraph_generators.HypergraphGenerators method*), 682
union() (*sage.graphs.generic_graph.GenericGraph method*), 217

UnitaryDualPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 522
UnitaryPolarGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 523
unlabeled_edges() (*sage.graphs.tutte_polynomial.Ear method*), 904
unpickle_graph_backend() (*in module sage.graphs.base.graph_backends*), 670
update_comp_num() (*in module sage.graphs.graph_decompositions.modular_decomposition*), 857
update_db() (*sage.graphs.isgci.GraphClasses method*), 599
USAMap() (*sage.graphs.graph_generators.GraphGenerators static method*), 522

V

vertex_boundary() (*sage.graphs.generic_graph.GenericGraph method*), 217
vertex_coloring() (*in module sage.graphs.graph_coloring*), 711
vertex_connectivity() (*in module sage.graphs.connectivity*), 953
vertex_connectivity() (*sage.graphs.generic_graph.GenericGraph method*), 218
vertex_cover() (*sage.graphs.bipartite_graph.BipartiteGraph method*), 395
vertex_cover() (*sage.graphs.graph.Graph method*), 339
vertex_cut() (*sage.graphs.generic_graph.GenericGraph method*), 220
vertex_disjoint_paths() (*sage.graphs.generic_graph.GenericGraph method*), 221
vertex_iterator() (*sage.graphs.generic_graph.GenericGraph method*), 221
vertex_separation() (*in module sage.graphs.graph_decompositions.vertex_separation*), 807
vertex_separation_BAB() (*in module sage.graphs.graph_decompositions.vertex_separation*), 808
vertex_separation_exp() (*in module sage.graphs.graph_decompositions.vertex_separation*), 811
vertex_separation_MILP() (*in module sage.graphs.graph_decompositions.vertex_separation*), 810
VertexOrder (*class in sage.graphs.tutte_polynomial*), 904
VertexPosition (*class in sage.graphs.graph_decompositions.modular_decomposition*), 827
vertices (*sage.graphs.tutte_polynomial.Ear attribute*), 904
vertices() (*sage.graphs.generic_graph.GenericGraph method*), 222
verts() (*sage.graphs.base.c_graph.CGraph method*), 614
verts() (*sage.graphs.base.static_sparse_backend.StaticSparseCGraph method*), 664

W

WagnerGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 523
WatkinsSnarkGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 523
weighted() (*sage.graphs.generic_graph.GenericGraph method*), 223
weighted_adjacency_matrix() (*sage.graphs.generic_graph.GenericGraph method*), 223
WellsGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 524
WheelGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 524
width_of_cut_decomposition() (*in module sage.graphs.graph_decompositions.cutwidth*), 821
width_of_path_decomposition() (*in module sage.graphs.graph_decompositions.vertex_separation*), 812
wiener_index() (*in module sage.graphs.distances_all_pairs*), 875
wiener_index() (*sage.graphs.generic_graph.GenericGraph method*), 224
WienerArayaGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 525
WindmillGraph() (*sage.graphs.graph_generators.GraphGenerators static method*), 525
WorldMap() (*sage.graphs.graph_generators.GraphGenerators static method*), 526
write_to_eps() (*sage.graphs.graph.Graph method*), 340

Y

yen_k_shortest_simple_paths() (*in module sage.graphs.path_enumeration*), 921