Sage Reference Manual: Graph Theory

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

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()	Creates a new NetworkX graph from the Sage graph
to_dictionary()	Creates a dictionary encoding the graph.
copy()	Return a copy of the graph.
adjacency_matrix()	Returns the adjacency matrix of the (di)graph.
incidence_matrix()	Returns an incidence matrix of the (di)graph
distance_matrix()	Returns the distance matrix of the (strongly) connected (di)graph
weighted_adjacency_matrix()	Returns the weighted adjacency matrix of the graph
kirchhoff_matrix()	Returns the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.
get_boundary()	Returns the boundary of the (di)graph.
set_boundary()	Sets the boundary of the (di)graph.
has_loops()	Returns whether there are loops in the (di)graph.
allows_loops()	Returns whether loops are permitted in the (di)graph.
allow_loops()	Changes whether loops are permitted in the (di)graph.
loops()	Returns any loops in the (di)graph.
has_multiple_edges()	Returns whether there are multiple edges in the (di)graph.
allows_multiple_edges()	Returns whether multiple edges are permitted in the (di)graph.
allow_multiple_edges()	Changes whether multiple edges are permitted in the (di)graph.
multiple_edges()	Returns any multiple edges in the (di)graph.
name()	Returns or sets the graph's name.
weighted()	Whether the (di)graph is to be considered as a weighted (di)graph.
antisymmetric()	Tests whether the graph is antisymmetric
density()	Returns the density
order()	Returns the number of vertices.
size()	Returns the number of edges.
add_vertex()	Creates an isolated vertex.
add_vertices()	Add vertices to the (di)graph from an iterable container
delete_vertex()	Deletes a vertex, removing all incident edges.
delete_vertices()	Remove vertices from the (di)graph taken from an iterable container of vertices.
has_vertex()	Return True if vertex is one of the vertices of this graph.
random_vertex()	Returns a random vertex of self.

Table 1.1 – continued from previous page

random_edge()	Returns a random edge of self.
vertex_boundary()	Returns a list of all vertices in the external boundary of vertices1, intersected with vertices
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.
loop_vertices()	Returns a list of vertices with loops.
vertex_iterator()	Returns an iterator over the 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()	Adds an edge from u and v.
add_edges()	Add edges from an iterable container.
subdivide_edge()	Subdivides an edge k times.
subdivide_edges()	Subdivides k times edges from an iterable container.
delete_edge()	Delete the edge from u to v
delete_edges()	Delete edges from an iterable container.
delete_multiedge()	Deletes all edges from u and v.
set_edge_label()	Set the edge label of a given edge.
has_edge()	Returns True if (u, v) is an edge, False otherwise.
edges()	Return a list of edges.
edge_boundary()	Returns a list of edges (u, v, l) with u in vertices1
edge_iterator()	Returns an iterator over edges.
edges_incident()	Returns incident edges to some vertices.
edge_label()	Returns the label of an edge.
edge_labels()	Returns a list of edge labels.
remove_multiple_edges()	Removes all multiple edges, retaining one edge for each.
remove_loops()	Removes loops on vertices in vertices. If vertices is None, removes all loops.
loop_edges()	Returns a list of all loops in the graph.
number_of_loops()	Returns the number of edges that are loops.
clear()	Empties the graph of vertices and edges and removes name, boundary, associated objects,
degree()	Gives the degree (in + out for digraphs) of a vertex or of vertices.
average_degree()	Returns the average degree of the graph.
degree_histogram()	Returns a list, whose ith entry is the frequency of degree i.
degree_iterator()	Returns 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 that contains each vertex with prob. p.
add_cycle()	Adds a cycle to the graph with the given vertices.
add_path()	Adds a cycle to the graph with the given vertices.
complement()	Returns the complement of the (di)graph.
line_graph()	Returns the line graph of the (di)graph.
to_simple()	Returns a simple version of itself (i.e., undirected and loops and multiple edges are remove
disjoint_union()	Returns the disjoint union of self and other.
union()	Returns the union of self and other.
relabel()	Relabels the vertices of self
degree_to_cell()	Returns the number of edges from vertex to an edge in cell.
subgraph()	Returns the subgraph containing the given vertices and edges.
is_subgraph()	Tests whether self is a subgraph of other.

Graph products:

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

Paths and cycles:

eulerian_orientation()	Returns 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.
cycle_basis()	Returns a list of cycles which form a basis of the cycle space of `self`.
interior_paths()	Returns an exhaustive list of paths (also lists) through only interior vertices from vertex start to vertex end in the (di)graph.
all_paths()	Returns a list of all paths (also lists) between a pair of vertices in the (di)graph.
triangles_count()	Returns the number of triangles in the (di)graph.

Linear algebra:

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

Some metrics:

cluster_triangles()	Returns the number of triangles for nbunch of vertices as a dictionary keyed by vertex.
clustering_average()	Returns the average clustering coefficient.
clustering_coeff()	Returns the clustering coefficient for each vertex in nbunch
cluster_transitivity()	Returns the transitivity (fraction of transitive triangles) of the graph.
szeged_index()	Returns the Szeged index of the graph.

Automorphism group:

coarsest_equitable_refinement()	Returns the coarsest partition which is finer than the input partition, and equitable with respect to self.
automorphism_group()	Returns the largest subgroup of the automorphism group of the (di)graph whose orbit partition is finer than the partition given.
is_vertex_transitive()	Returns whether the automorphism group of self is transitive within the partition provided
is_isomorphic()	Tests for isomorphism between self and other.
canonical_label()	Returns the unique graph on $\{0, 1, ..., n-1\}$ (`n = self.order()`) which 1) is isomorphic to self 2) is invariant in the isomorphism class.

Graph properties:

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

Traversals:

breadth_first_search()	Returns an iterator over the vertices in a breadth-first ordering.
depth_first_search()	Returns an iterator over the vertices in a depth-first ordering.
lex_BFS()	Performs a Lex BFS on the graph.

Distances:

distance()	Returns the (directed) distance from u to v in the (di)graph
distance_all_pairs()	Returns the distances between all pairs of vertices.
distances_distribution()	Returns the distances distribution of the (di)graph in a dictionary.
eccentricity()	Return the eccentricity of vertex (or vertices) v.
radius()	Returns the radius of the (di)graph.
center()	Returns the set of vertices in the center of the graph
diameter()	Returns the largest distance between any two vertices.
distance_graph()	Returns 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()	Computes the girth of the graph.
periphery()	Returns the set of vertices in the periphery
shortest_path()	Returns a list of vertices representing some shortest path from u to v
shortest_path_length()	Returns the minimal length of paths from u to v
shortest_paths()	Returns a dictionary associating to each vertex v a shortest path from u to v, if it exists.
shortest_path_lengths()	Returns a dictionary of shortest path lengths keyed by targets that are connected by a path from u.
shortest_path_all_pairs()	Computes a shortest path between each pair of vertices.
wiener_index()	Returns the Wiener index of the graph.
average_distance()	Returns the average distance between vertices of the graph.

Flows, connectivity, trees:

is_connected()	Tests whether the (di)graph is connected.
connected_components()	Returns the list of connected components
connected_components_number()	Returns the number of connected components.
connected_components_subgraphs()	Returns a list of connected components as graph objects.
connected_component_containing_vertex()	Returns a list of the vertices connected to vertex.
blocks_and_cut_vertices()	Computes the blocks and cut vertices of the graph.
blocks_and_cuts_tree()	Computes the blocks-and-cuts tree of the graph.
is_cut_edge()	Returns True if the input edge is a cut-edge or a bridge.
is_cut_vertex()	Returns True if the input vertex is a cut-vertex.
edge_cut()	Returns a minimum edge cut between vertices s and t
vertex_cut()	Returns a minimum vertex cut between non-adjacent vertices s and t
flow()	Returns a maximum flow in the graph from x to y
edge_disjoint_paths()	Returns a list of edge-disjoint paths between two vertices
vertex_disjoint_paths()	Returns a list of vertex-disjoint paths between two vertices as given by Menger's theorem.
edge_connectivity()	Returns the edge connectivity of the graph.
vertex_connectivity()	Returns the vertex connectivity of the graph.
transitive_closure()	Computes the transitive closure of a graph and returns it.
transitive_reduction()	Returns a transitive reduction of a graph.
min_spanning_tree()	Returns the edges of a minimum spanning tree.
spanning_trees_count()	Returns the number of spanning trees in a graph.

Plot/embedding-related methods:

set_embedding()	Sets a combinatorial embedding dictionary to _embedding attribute.
get_embedding()	Returns the attribute _embedding if it exists.
faces()	Returns the faces of an embedded graph.
get_pos()	Returns the position dictionary
set_pos()	Sets the position dictionary.
set_planar_positions()	Compute a planar layout for self using Schnyder's algorithm
layout_planar()	Uses Schnyder's algorithm to compute a planar layout for self.
is_drawn_free_of_edge_crossings()	Tests whether the position dictionary gives a planar embedding.
latex_options()	Returns an instance of GraphLatex for the graph.
set_latex_options()	Sets multiple options for rendering a graph with LaTeX.
layout()	Returns a layout for the vertices of this graph.
layout_spring()	Computes a spring layout for this graph
layout_ranked()	Computes a ranked layout for this graph
layout_extend_randomly()	Extends randomly a partial layout
layout_circular()	Computes a circular layout for this graph
layout_tree()	Computes an ordered tree layout for this graph, which should be a tree (no non-oriented cycles).
layout_graphviz()	Calls graphviz to compute a layout of the vertices of this graph.
graphplot()	Returns a GraphPlot object.
plot()	Returns a graphics object representing the (di)graph.
show()	Shows the (di)graph.
plot3d()	Plot a graph in three dimensions.
show3d()	Plots the graph using Tachyon, and shows the resulting plot.
graphviz_string()	Returns a representation in the dot language.
graphviz_to_file_named()	Write a representation in the dot in a file.

Algorithmically hard stuff:

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

add_cycle(*vertices*)

Adds 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` – a list of indices for 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(range(20)[10:20])
sage: show(G)
sage: G.add_cycle(range(10))
sage: show(G)

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

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

Adds an edge from u and 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.networkx_graph().adj           # random output order
{'label': {(1, 2): None}, (1, 2): {'label': None}}
```

Use one of these instead:

```
sage: G = Graph()
sage: G.add_edge((1,2), label="label")
sage: G.networkx_graph().adj           # random output order
{1: {2: 'label'}, 2: {1: 'label'}}

sage: G = Graph()
sage: G.add_edge(1,2,'label')
sage: G.networkx_graph().adj           # random output order
{1: {2: 'label'}, 2: {1: 'label'}}
```

The following syntax is supported, but note that you must use the `label` keyword:

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

Vertex name cannot be None, so:

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

add_edges(*edges*)

Add edges from an iterable container.

All elements of `edges` must follow the same format, i.e. have the same length.

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)])
sage: H.edges()
[(0, 1, None), (0, 2, None)]
```

add_path(*vertices*)

Adds 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 path vertices[0], ..., vertices[-1].

INPUT:

- `vertices` - a list of indices for 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_path(range(20)[10:20])
sage: show(G)
sage: G.add_path(range(10))
sage: show(G)
```

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

add_vertex(*name=None*)

Creates an isolated vertex. If the vertex already exists, then nothing is done.

INPUT:

- `name` - Name of the new vertex. If no name is specified, then the vertex will be represented by the least integer not already representing a vertex. Name must be an immutable object, and cannot be None.

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``=``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 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).vertices())
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*, *boundary_first=False*)

Returns the adjacency matrix of the (di)graph.

Each vertex is represented by its position in the list returned by the vertices() function.

The matrix returned is over the integers. If a different ring is desired, use either the change_ring function or the matrix function.

INPUT:

- `sparse` - whether to represent with a sparse matrix
- `boundary_first` - whether to represent the boundary vertices in the upper left block

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]
```

TESTS:

```
sage: graphs.CubeGraph(8).adjacency_matrix().parent()
Full MatrixSpace of 256 by 256 dense matrices over Integer Ring
sage: graphs.CubeGraph(9).adjacency_matrix().parent()
Full MatrixSpace of 512 by 512 sparse matrices over Integer Ring
sage: Graph([(i,i+1) for i in range(500)]+[(0,1),], multiedges=True).adjacency_matrix().pare
Full MatrixSpace of 501 by 501 dense matrices over Integer Ring
```

all_paths(*start*, *end*)

Returns a list of all paths (also lists) between a pair of vertices (start, end) in the (di)graph. If `start` is the same vertex as `end`, then `[[start]]` is returned – a list containing the 1-vertex, 0-edge path "`start`".

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]]
```

Starting and ending at the same vertex (see trac ticket #13006):

```
sage: graphs.CompleteGraph(4).all_paths(2,2)
[[2]]
```

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

Changes whether loops are permitted in the (di)graph.

INPUT:

- `new` - boolean.

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()
[]
```

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

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

INPUT:

- `new` - boolean.

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

allows_loops()

Returns 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()
[]

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()

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

am(*sparse=None*, *boundary_first=False*)

Returns the adjacency matrix of the (di)graph.

Each vertex is represented by its position in the list returned by the vertices() function.

The matrix returned is over the integers. If a different ring is desired, use either the change_ring function or the matrix function.

INPUT:

- `sparse` - whether to represent with a sparse matrix
- `boundary_first` - whether to represent the boundary vertices in the upper left block

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]
```

TESTS:

```
sage: graphs.CubeGraph(8).adjacency_matrix().parent()
Full MatrixSpace of 256 by 256 dense matrices over Integer Ring
```

```
sage: graphs.CubeGraph(9).adjacency_matrix().parent()
Full MatrixSpace of 512 by 512 sparse matrices over Integer Ring
sage: Graph([(i,i+1) for i in range(500)]+[(0,1),], multiedges=True).adjacency_matrix().pare
Full MatrixSpace of 501 by 501 dense matrices over Integer Ring
```

antisymmetric()

Tests whether the graph is antisymmetric.

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

A directed acyclic graph is antisymmetric. An undirected graph is never antisymmetric unless it is just a union of isolated vertices.

```
sage: graphs.RandomGNP(20,0.5).antisymmetric()
False
sage: digraphs.RandomDirectedGNR(20,0.5).antisymmetric()
True
```

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

Returns the largest subgroup of the automorphism group of the (di)graph whose orbit partition is finer than the partition given. If no partition is given, the unit partition is used and the entire automorphism group is 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

Warning: Since trac ticket #14319 the domain of the automorphism group is equal to the graph's vertex set, and the `translation` argument has become useless.

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'))
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 = 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(implementation='c_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)
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
sage: G.automorphism_group(partition=[[0],range(1,10)], return_group=False, orbits=True)
[[0], [2, 3, 6, 7, 8, 9], [1, 4, 5]]
sage: C = graphs.CubeGraph(3)
sage: C.automorphism_group(orbits=True, return_group=False)
[['000', '001', '010', '011', '100', '101', '110', '111']]
```

TESTS:

We get a KeyError when given an invalid partition (trac #6087):

```
sage: g=graphs.CubeGraph(3)
sage: g.relabel()
sage: g.automorphism_group(partition=[[0,1,2],[3,4,5]])
Traceback (most recent call last):
...
KeyError: 6
```

Labeled automorphism group:

```
sage: digraphs.DeBruijn(3,2).automorphism_group()
Permutation Group with generators [('01','02')('10','20')('11','22')('12','21'), ('00','11')
sage: d = digraphs.DeBruijn(3,2)
sage: d.allow_multiple_edges(True)
sage: d.add_edge(d.edges()[0])
sage: d.automorphism_group()
Permutation Group with generators [('01','02')('10','20')('11','22')('12','21')]
```

The labeling is correct:

```
sage: g = graphs.PetersenGraph()
sage: ag = g.automorphism_group()
sage: for u,v in g.edges(labels = False):
...       if len(ag.orbit((u,v),action="OnPairs")) != 30:
...           print "ARgggggggggggg !!!"
```

Empty group, correct domain:

```
sage: Graph({'a':['a'], 'b':[]}).automorphism_group()
Permutation Group with generators [()]
sage: Graph({'a':['a'], 'b':[]}).automorphism_group().domain()
{'a', 'b'}
```

We can check that the subgroups are labelled correctly (trac ticket #15656):

```
sage: G1 = Graph(':H`ECw@HGXGAGUG`e')
sage: G = G1.automorphism_group()
sage: G.subgroups()
[Subgroup of (Permutation Group with generators [(0,7)(1,4)(2,3)(6,8)]) generated by [()],
Subgroup of (Permutation Group with generators [(0,7)(1,4)(2,3)(6,8)]) generated by [(0,7)(1
```

average_degree()

Returns 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: g = graphs.RandomGNP(20,.5)
sage: tree = Graph()
sage: tree.add_edges(g.min_spanning_tree())
sage: tree.average_degree() < 2
True
```

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

```
sage: g = graphs.RandomGNP(50,.8)
sage: g.average_degree() == 2*g.size()/g.order()
True
```

average_distance()

Returns 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.

EXAMPLE:

From [GYLL93]:

```
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
```

REFERENCE:

TEST:

```
sage: g = Graph()
sage: g.average_distance()
Traceback (most recent call last):
...
ValueError: The graph must have at least two vertices for this value to be defined
```

blocks_and_cut_vertices()

Computes 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.

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()`

EXAMPLES:

```
sage: graphs.PetersenGraph().blocks_and_cut_vertices()
([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]], [])
sage: graphs.PathGraph(6).blocks_and_cut_vertices()
([[4, 5], [3, 4], [2, 3], [1, 2], [0, 1]], [1, 2, 3, 4])
sage: graphs.CycleGraph(7).blocks_and_cut_vertices()
([[0, 1, 2, 3, 4, 5, 6]], [])
sage: graphs.KrackhardtKiteGraph().blocks_and_cut_vertices()
([[8, 9], [7, 8], [0, 1, 2, 3, 4, 5, 6, 7]], [7, 8])
sage: G=Graph()  # make a bowtie graph where 0 is a cut vertex
sage: G.add_vertices(range(5))
sage: G.add_edges([(0,1),(0,2),(0,3),(0,4),(1,2),(3,4)])
sage: G.blocks_and_cut_vertices()
([[0, 1, 2], [0, 3, 4]], [0])
sage: graphs.StarGraph(3).blocks_and_cut_vertices()
([[0, 1], [0, 2], [0, 3]], [0])
```

TESTS:

```
sage: Graph(0).blocks_and_cut_vertices()
([], [])
sage: Graph(1).blocks_and_cut_vertices()
([[0]], [])
sage: Graph(2).blocks_and_cut_vertices()
Traceback (most recent call last):
...
NotImplementedError: ...
```

REFERENCE:

blocks_and_cuts_tree()

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

See Also:

`blocks_and_cut_vertices()`

EXAMPLES:

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

The distance between two leaves is even:

```
sage: T = graphs.RandomTree(40).blocks_and_cuts_tree()
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 = graphs.PetersenGraph().blocks_and_cuts_tree()
sage: T.vertices()
[('B', (0, 1, 2, 3, 4, 5, 6, 7, 8, 9))]
```

REFERENCES:

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

Returns 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` - (default False) only applies to directed graphs. If True, searches across edges in either direction.
- `distance` - the maximum distance from the `start` nodes to traverse. The `start` nodes are distance zero from themselves.
- `neighbors` - a function giving the neighbors of a vertex. The function should take a vertex and return a list of vertices. For a graph, `neighbors` is by default the `neighbors()` function of the graph. For a digraph, the `neighbors` function defaults to the `successors()` function of the graph.

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]
```

TESTS:

```
sage: D = DiGraph({1:[0], 2:[0]})
sage: list(D.breadth_first_search(0))
[0]
sage: list(D.breadth_first_search(0, ignore_direction=True))
[0, 1, 2]
```

canonical_label (*partition=None*, *certify=False*, *verbosity=0*, *edge_labels=False*)

Returns the unique graph on $\{0, 1, ..., n-1\}$ (`n = self.order()`) which

- is isomorphic to self,
- is invariant in the isomorphism class.

In other words, given two graphs `G` and `H` which are isomorphic, suppose `G_c` and `H_c` are the graphs returned by `canonical_label`. Then the following hold:

- `G_c == H_c`
- `G_c.adjacency_matrix() == H_c.adjacency_matrix()`
- `G_c.graph6_string() == H_c.graph6_string()`

INPUT:

- `partition` - if given, the canonical label with respect to this set partition will be computed. The default is the unit set partition.
- `certify` - if True, a dictionary mapping from the (di)graph to its canonical label will be given.
- `verbosity` - gets passed to nice: prints helpful output.
- `edge_labels` - default False, otherwise allows only permutations respecting edge labels.

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: E = D.canonical_label(); E
Dodecahedron: Graph on 20 vertices
sage: D.canonical_label(certify=True)
(Dodecahedron: Graph on 20 vertices, {0: 0, 1: 19, 2: 16, 3: 15, 4: 9, 5: 1, 6: 10, 7: 8, 8:
sage: D.is_isomorphic(E)
True
```

Multigraphs:

```
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?', implementation='c_graph').canonical_label()
Graph on 2 vertices
```

Digraphs:

```
sage: P = graphs.PetersenGraph()
sage: DP = P.to_directed()
sage: DP.canonical_label().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,certify=True)
(Graph on 5 vertices, {0: 4, 1: 3, 2: 0, 3: 1, 4: 2})
```

Check for immutable graphs (trac ticket #16602):

```
sage: G = Graph([[1, 2], [2, 3]], immutable=True)
sage: C = G.canonical_label(); C
Graph on 3 vertices
sage: C.vertices()
[0, 1, 2]
```

cartesian_product(*other*)

Returns 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.

TESTS:

Cartesian product of graphs:

```
sage: G = Graph([(0,1),(1,2)])
sage: H = Graph([('a','b')])
sage: C1 = G.cartesian_product(H)
sage: C1.edges(labels=None)
[((0, 'a'), (0, 'b')), ((0, 'a'), (1, 'a')), ((0, 'b'), (1, 'b')), ((1, 'a'), (1, 'b')), ((1
sage: C2 = H.cartesian_product(G)
sage: C1.is_isomorphic(C2)
True
```

Construction of a Toroidal grid:

```
sage: A = graphs.CycleGraph(3)
sage: B = graphs.CycleGraph(4)
sage: T = A.cartesian_product(B)
sage: T.is_isomorphic( graphs.ToroidalGrid2dGraph(3,4) )
True
```

Cartesian product of digraphs:

```
sage: P = DiGraph([(0,1)])
sage: B = digraphs.DeBruijn( ['a','b'], 2 )
sage: Q = P.cartesian_product(B)
sage: Q.edges(labels=None)
[((0, 'aa'), (0, 'aa')), ((0, 'aa'), (0, 'ab')), ((0, 'aa'), (1, 'aa')), ((0, 'ab'), (0, 'ba
sage: Q.strongly_connected_components_digraph().num_verts()
2
sage: V = Q.strongly_connected_component_containing_vertex( (0, 'aa') )
sage: B.is_isomorphic( Q.subgraph(V) )
True
```

categorical_product(*other*)

Returns 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 Wikipedia article on the 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
```

```
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
```

TESTS:

Tensor product of graphs:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.tensor_product(H)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((0, 'b'), (1, 'a')), ((1, 'a'), (2, 'b')), ((1, 'b'), (2, 'a'))]
sage: T.is_isomorphic( H.tensor_product(G) )
True
```

Tensor product of digraphs:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.tensor_product(J)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((1, 'a'), (2, 'b'))]
sage: T.is_isomorphic( J.tensor_product(I) )
True
```

The tensor product of two DeBruijn digraphs of same diameter is a DeBruijn digraph:

```
sage: B1 = digraphs.DeBruijn(2, 3)
sage: B2 = digraphs.DeBruijn(3, 3)
sage: T = B1.tensor_product( B2 )
sage: T.is_isomorphic( digraphs.DeBruijn( 2*3, 3) )
True
```

center()

Returns the set of vertices in the center, i.e. whose eccentricity is equal to the radius of the (di)graph.

In other words, the center is the set of vertices achieving the minimum eccentricity.

EXAMPLES:

```
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]
sage: G = Graph()
sage: G.center()
[]
sage: G.add_vertex()
0
sage: G.center()
[0]
```

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

Returns 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` – (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
```

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

Returns 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` – (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
```

check_embedding_validity(**args*, ***kwds*)
Deprecated: Use `_check_embedding_validity()` instead. See trac ticket #15551 for details.

check_pos_validity(**args*, ***kwds*)
Deprecated: Use `_check_pos_validity()` instead. See trac ticket #15551 for details.

clear()
Empties the graph of vertices and edges and removes name, boundary, 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(implementation='c_graph', sparse=True)
sage: H.clear()
sage: H.order(); H.size()
0
0
sage: len(H._pos)
0
sage: H.name()
''
sage: H.get_vertex(0)
sage: H = G.copy(implementation='c_graph', 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)
sage: H = G.copy(implementation='networkx')
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()
Returns the transitivity (fraction of transitive triangles) of the graph.

Transitivity is the fraction of all existing triangles and all connected triples (triads), $T = 3 \times$ triangles/triads.

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

EXAMPLES:

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

cluster_triangles(*nbunch=None*, *with_labels=False*)

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

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

INPUT:

- `nbunch` - The vertices to inspect. If `nbunch=None`, returns data for all vertices in the graph.

REFERENCE:

EXAMPLES:

```
sage: (graphs.FruchtGraph()).cluster_triangles().values()
[1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0]
sage: (graphs.FruchtGraph()).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: (graphs.FruchtGraph()).cluster_triangles(nbunch=[0,1,2])
{0: 1, 1: 1, 2: 0}
```

clustering_average()

Returns the average clustering coefficient.

The clustering coefficient of a node i is the fraction of existing triangles containing node i and 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 [HSSNX].

EXAMPLES:

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

clustering_coeff(*nodes=None*, *weight=False*, *return_vertex_weights=True*)

Returns 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 and 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 [HSSNX].

INPUT:

- `nodes` - the vertices to inspect (default `None`, returns data on all vertices in graph)
- `weight` - string or boolean (default is `False`). If it is a string it used the indicated edge property as weight. `weight = True` is equivalent to `weight = 'weight'`

- `return_vertex_weights` is a boolean ensuring backwards compatibility with deprecated features of NetworkX 1.2. It should be set to `False` for all production code.

EXAMPLES:

```
sage: (graphs.FruchtGraph()).clustering_coeff().values()
[0.3333333333333333, 0.3333333333333333, 0.0, 0.3333333333333333,
 0.3333333333333333, 0.3333333333333333, 0.3333333333333333,
 0.3333333333333333, 0.0, 0.3333333333333333, 0.3333333333333333,
 0.0]
sage: (graphs.FruchtGraph()).clustering_coeff()
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.0,
 3: 0.3333333333333333, 4: 0.3333333333333333,
 5: 0.3333333333333333, 6: 0.3333333333333333,
 7: 0.3333333333333333, 8: 0.0, 9: 0.3333333333333333,
 10: 0.3333333333333333, 11: 0.0}

sage: (graphs.FruchtGraph()).clustering_coeff(weight=True,
...      return_vertex_weights=False)
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.0,
3: 0.3333333333333333, 4: 0.3333333333333333,
5: 0.3333333333333333, 6: 0.3333333333333333,
7: 0.3333333333333333, 8: 0.0, 9: 0.3333333333333333,
10: 0.3333333333333333, 11: 0.0}
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, return_vertex_weights=False)
{0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.0}
```

TESTS:

Doctests that demonstrate the deprecation of the two-dictionary return value due to the NetworkX API change after 1.2. The return_vertex_weights keyword is provided with a default value of True for backwards compatibility with older versions of Sage. When the deprecation period has expired and the keyword is removed, these doctests should be removed as well.

```
sage: (graphs.FruchtGraph()).clustering_coeff(weight=True,
...      return_vertex_weights=True)
doctest:...: DeprecationWarning: The option 'return_vertex_weights'
has been deprecated. Only offered for backwards compatibility with
NetworkX 1.2.
See http://trac.sagemath.org/12806 for details.
({0: 0.3333333333333333, 1: 0.3333333333333333, 2: 0.0,
  3: 0.3333333333333333, 4: 0.3333333333333333,
  5: 0.3333333333333333, 6: 0.3333333333333333,
  7: 0.3333333333333333, 8: 0.0, 9: 0.3333333333333333,
  10: 0.3333333333333333, 11: 0.0}, {0: 0.08333333333333333,
  1: 0.08333333333333333, 2: 0.08333333333333333,
  3: 0.08333333333333333, 4: 0.08333333333333333,
  5: 0.08333333333333333, 6: 0.08333333333333333,
  7: 0.08333333333333333, 8: 0.08333333333333333,
  9: 0.08333333333333333, 10: 0.08333333333333333,
  11: 0.08333333333333333})

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

```
 2: 0.3333333333333333})
```

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

Returns 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 C1, C2 of the partition, the number of edges from a vertex of C1 to C2 is the same, over all vertices in C1.

A partition P1 is finer than P2 (P2 is coarser than P1) if every cell of P1 is a subset of a cell of P2.

INPUT:

- `partition` - a list of lists
- `sparse` - (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],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: 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)]]

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

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

complement()

Returns 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
sage: PC = P.complement()
sage: PC.plot() # long time
```

```
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):
...
TypeError: complement not well defined for (di)graphs with multiple edges
```

TESTS:

We check that trac ticket #15699 is fixed:

```
sage: G = graphs.PathGraph(5).copy(immutable=True)
sage: G.complement()
complement(Path Graph): Graph on 5 vertices
```

connected_component_containing_vertex(*vertex*)

Returns a list of the vertices connected to vertex.

EXAMPLES:

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

connected_components()

Returns the list of connected components.

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

EXAMPLES:

```
sage: G = Graph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 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: D.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
```

connected_components_number()

Returns the number of connected components.

EXAMPLES:

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

connected_components_subgraphs()

Returns a list of connected components as graph objects.

EXAMPLES:

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

copy(*implementation='c_graph'*, *data_structure=None*, *sparse=None*, *immutable=None*)

Return a copy of the graph.

INPUT:

- `implementation` - string (default: 'c_graph') the implementation goes here. Current options are only 'networkx' or 'c_graph'.
- `sparse` (boolean) – `sparse=True` is an alias for `data_structure="sparse"`, and `sparse=False` is an alias for `data_structure="dense"`. Only used when `implementation='c_graph'` and `data_structure=None`.
- `data_structure` – one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of `Graph` or `DiGraph`. Only used when `implementation='c_graph'`.
- `immutable` (boolean) – whether to create a mutable/immutable copy. Only used when `implementation='c_graph'` and `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'` and `implementation='c_graph'`
  - `immutable=False` sets `implementation` to `'c_graph'`. When `immutable=False` is 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 implementation. Otherwise simply do `copy(g)` instead of `g.copy()`.

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.MoebiusKanto
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(19)
sage: H = G.copy(implementation='c_graph')
sage: H == G; H is G
True
False
sage: G1 = G.copy(sparse=True)
sage: G1==G
True
sage: G1 is G
False
sage: G2 = copy(G)
sage: G2 is G
False
```

TESTS:

We make copies of the `_pos` and `_boundary` attributes:

```
sage: g = graphs.PathGraph(3)
sage: h = copy(g)
sage: h._pos is g._pos
False
sage: h._boundary is g._boundary
False
```

We make sure that one can make immutable copies by providing the `data_structure` optional argument, and that copying an immutable graph returns the graph:

```
sage: G = graphs.PetersenGraph()
sage: hash(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 = G.copy(immutable=True)
sage: hash(g)    # random
1833517720
sage: g==G
True
sage: g is copy(g) is g.copy()
True
```

`immutable=True` is a short-cut for `data_structure='static_sparse'`:

```
sage: g is g.copy(data_structure='static_sparse') is g.copy(immutable=True)
True
```

If a graph pretends to be immutable, but does not use the static sparse backend, then the copy is not identic with the graph, even though it is considered to be hashable:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True, facade = False)
sage: H = P.hasse_diagram()
sage: H._immutable = True
sage: hash(H)    # random
-1843552882
sage: copy(H) is H
False
```

TESTS:

Bad input:

```
sage: G.copy(data_structure="sparse", sparse=False)
Traceback (most recent call last):
...
ValueError: You cannot define 'immutable' or 'sparse' when 'data_structure' has a value.
sage: G.copy(data_structure="sparse", immutable=True)
Traceback (most recent call last):
...
ValueError: You cannot define 'immutable' or 'sparse' when 'data_structure' has a value.
sage: G.copy(immutable=True, sparse=False)
Traceback (most recent call last):
...
ValueError: There is no dense immutable backend at the moment.
```

Which backend ?:

```
sage: G.copy(data_structure="sparse")._backend
<class 'sage.graphs.base.sparse_graph.SparseGraphBackend'>
sage: G.copy(data_structure="dense")._backend
<class 'sage.graphs.base.dense_graph.DenseGraphBackend'>
sage: G.copy(data_structure="static_sparse")._backend
<class 'sage.graphs.base.static_sparse_backend.StaticSparseBackend'>
sage: G.copy(immutable=True)._backend
<class 'sage.graphs.base.static_sparse_backend.StaticSparseBackend'>
sage: G.copy(immutable=True, sparse=True)._backend
<class 'sage.graphs.base.static_sparse_backend.StaticSparseBackend'>
sage: G.copy(immutable=False, sparse=True)._backend
<class 'sage.graphs.base.sparse_graph.SparseGraphBackend'>
sage: G.copy(immutable=False, sparse=False)._backend
<class 'sage.graphs.base.sparse_graph.SparseGraphBackend'>
sage: Graph(implementation="networkx").copy(implementation='c_graph')._backend
<class 'sage.graphs.base.sparse_graph.SparseGraphBackend'>
```

Fake immutable graphs:

```
sage: G._immutable = True
sage: G.copy()._backend
<class 'sage.graphs.base.sparse_graph.SparseGraphBackend'>
```

cycle_basis(*output='vertex'*)

Returns 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.

INPUT:

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

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.

EXAMPLE:

A cycle basis in Petersen's Graph

```
sage: g = graphs.PetersenGraph()
sage: g.cycle_basis()
[[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,
```

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

```
sage: g.cycle_basis(output='edge')
[[(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)]]
```

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 an 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), edge_space.basis() ) )
sage: for (u,v),vec in 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 = map( cycle_to_vector, 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')])
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')]]
```

Disconnected graph:

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

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 with multiple edges:

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

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

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

INPUT:

- `vertices` - If vertices is a single vertex, returns the number of neighbors of vertex. If vertices is an iterable container of vertices, returns a list of degrees. If vertices is None, same as listing all vertices.
- `labels` - see OUTPUT

OUTPUT: Single vertex- an integer. Multiple vertices- a list of integers. If labels is True, then returns a dictionary mapping each vertex 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]
```

degree_histogram()

Returns a list, whose ith 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*)

Returns 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:

- `labels` (boolean) – if set to `False` (default) the method returns an iterator over degrees. Otherwise it returns an iterator over tuples (vertex, degree).
- `vertices` - if specified, restrict to this subset.

EXAMPLES:

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

sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: for i in D.degree_iterator():
...     print i
6
6
...
4
6
sage: for i in D.degree_iterator(labels=True):
...     print i
((0, 1), 6)
((1, 2), 6)
...
((0, 3), 4)
((1, 1), 6)
```

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, returning 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(19).copy(implementation='c_graph')
sage: G.size()
171
sage: G.delete_edge( 1, 2 )
sage: G.delete_edge( (3, 4) )
sage: G.delete_edges( [ (5, 6), (7, 8) ] )
sage: G.size()
167
```

Note that NetworkX accidentally deletes these edges, even though the labels do not match up:

```
sage: N = graphs.CompleteGraph(19).copy(implementation='networkx')
sage: N.size()
171
sage: N.delete_edge( 1, 2 )
sage: N.delete_edge( (3, 4) )
sage: N.delete_edges( [ (5, 6), (7, 8) ] )
sage: N.size()
167
sage: N.delete_edge( 9, 10, 'label' )
sage: N.delete_edge( (11, 12, 'label') )
sage: N.delete_edges( [ (13, 14, 'label') ] )
sage: N.size()
167
sage: N.has_edge( (11, 12) )
True
```

However, CGraph backends handle things properly:

```
sage: G.delete_edge( 9, 10, 'label' )
sage: G.delete_edge( (11, 12, 'label') )
sage: G.delete_edges( [ (13, 14, 'label') ] )
sage: G.size()
167

sage: C = graphs.CompleteGraph(19).to_directed(sparse=True)
sage: C.size()
342
sage: C.delete_edge( 1, 2 )
sage: C.delete_edge( (3, 4) )
sage: C.delete_edges( [ (5, 6), (7, 8) ] )

sage: D = graphs.CompleteGraph(19).to_directed(sparse=True, implementation='networkx')
sage: D.size()
342
sage: D.delete_edge( 1, 2 )
sage: D.delete_edge( (3, 4) )
sage: D.delete_edges( [ (5, 6), (7, 8) ] )
sage: D.delete_edge( 9, 10, 'label' )
sage: D.delete_edge( (11, 12, 'label') )
sage: D.delete_edges( [ (13, 14, 'label') ] )
sage: D.size()
338
sage: D.has_edge( (11, 12) )
True

sage: C.delete_edge( 9, 10, 'label' )
sage: C.delete_edge( (11, 12, 'label') )
sage: C.delete_edges( [ (13, 14, 'label') ] )
```

```
sage: C.size() # correct!
338
sage: C.has_edge( (11, 12) ) # correct!
True
```

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 = graphs.CompleteGraph(12).to_directed()
sage: K4 = graphs.CompleteGraph(4).to_directed()
sage: K12.size()
132
sage: K12.delete_edges(K4.edge_iterator())
sage: K12.size()
120
```

delete_multiedge(*u*, *v*)

Deletes all edges from u and v.

EXAMPLES:

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

Deletes vertex, removing all incident edges. Deleting a non-existent vertex will raise an exception.

INPUT:

- `in_order` - (default False) If True, this deletes the ith 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):
...
RuntimeError: 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.set_boundary([1,2])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: G.delete_vertex(1)
sage: G.get_vertices()
{0: 'no delete', 2: None, 3: None, 4: None}
sage: G.get_boundary()
[2]
sage: G.get_pos()
{0: (0, 0), 2: (2, 0), 3: (3, 0), 4: (4, 0)}
```

delete_vertices(*vertices*)

Remove vertices from the (di)graph taken from an iterable container of vertices. Deleting a non-existent vertex will raise an exception.

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):
...
RuntimeError: Vertex (1) not in the graph.
```

density()

Returns the density (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*, *distance=None*, *neighbors=None*)

Returns 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` - (default False) only applies to directed graphs. If True, searches across edges in either direction.
- `distance` - the maximum distance from the `start` nodes to traverse. The `start` nodes are distance zero from themselves.
- `neighbors` - a function giving the neighbors of a vertex. The function should take a vertex and return a list of vertices. For a graph, `neighbors` is by default the `neighbors()` function of the graph. For a digraph, the `neighbors` function defaults to the `successors()` function of the graph.

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]
```

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.depth_first_search(0,distance=0))
[0]
sage: list(D.depth_first_search(0,distance=1))
[0, 3, 2, 1]
```

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]
sage: list(D.depth_first_search([0,6],distance=0))
```

```
[0, 6]
sage: list(D.depth_first_search([0,6],distance=1))
[0, 3, 2, 1, 6, 7]
sage: list(D.depth_first_search(6,ignore_direction=True,distance=2))
[6, 7, 5, 0, 3]
```

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.depth_first_search(5,neighbors=D.neighbors_in, distance=2))
[5, 2, 0, 1]
sage: list(D.depth_first_search(5,neighbors=D.neighbors_out, distance=2))
[5, 7, 0]
sage: list(D.depth_first_search(5,neighbors=D.neighbors, distance=2))
[5, 7, 6, 0, 2, 1, 4]
```

TESTS:

```
sage: D = DiGraph({1:[0], 2:[0]})
sage: list(D.depth_first_search(0))
[0]
sage: list(D.depth_first_search(0, ignore_direction=True))
[0, 2, 1]
```

diameter()

Returns the largest distance between any two vertices. Returns Infinity if the (di)graph is not connected.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.diameter()
2
sage: G = Graph( { 0 : [], 1 : [], 2 : [1] } )
sage: G.diameter()
+Infinity
```

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: G.diameter()
0
```

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

Returns 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 interset at their endpoints and are vertex-disjoint otherwise.

INPUT:

- `pairs` – list of pairs of vertices
- `solver` – Specify a Linear Program solver to be used. If set to `None`, the default one is used. function of `MixedIntegerLinearProgram`. See the documentation of `MixedIntegerLinearProgram.solve` for more informations.
- `verbose` (integer) – sets the level of verbosity. Set to 0 by default (quiet).

EXAMPLE:

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.GridGraph([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.GridGraph([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*, *verbose_relabel=True*)

Returns the disjoint union of self and other.

INPUT:

- verbose_relabel - (defaults to True) If True, each vertex v in the first graph will be named '0,v' and each vertex u in the second graph will be named '1,u' in the final graph. If False, the vertices of the first graph and the second graph will be relabeled with consecutive integers.

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, verbose_relabel=False); J
Cycle graph disjoint_union Cycle graph: Graph on 7 vertices
sage: J.vertices()
[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*)

Returns 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

sage: C = graphs.CycleGraph(5)
sage: D = C.disjunctive_product(Z); D
Graph on 10 vertices
sage: D.plot() # long time
```

TESTS:

Disjunctive product of graphs:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.disjunctive_product(H)
sage: T.edges(labels=None)
[((0, 'a'), (0, 'b')), ((0, 'a'), (1, 'a')), ((0, 'a'), (1, 'b')), ((0, 'a'), (2, 'b')), ((0
sage: T.is_isomorphic( H.disjunctive_product(G) )
True
```

Disjunctive product of digraphs:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.disjunctive_product(J)
sage: T.edges(labels=None)
[((0, 'a'), (0, 'b')), ((0, 'a'), (1, 'a')), ((0, 'a'), (1, 'b')), ((0, 'a'), (2, 'b')), ((0
sage: T.is_isomorphic( J.disjunctive_product(I) )
True
```

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

Returns the (directed) distance from u to v in the (di)graph, i.e. the length of the shortest path from u to v.

INPUT:

- `by_weight` - if `False`, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.

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
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(*algorithm='auto'*)

Returns the distances between all pairs of vertices.

INPUT:

- "algorithm" (string) – two algorithms are available
  - algorithm = "BFS" in which case the distances are computed through n different breadth-first-search.
  - algorithm = "Floyd-Warshall", in which case the Floyd-Warshall algorithm is used.
  - algorithm = "auto", in which case the Floyd-Warshall algorithm is used for graphs on less than 20 vertices, and BFS otherwise.

  The default is algorithm = "BFS".

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.

EXAMPLE:

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
```

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()

distance_graph(*dist*)

Returns 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 is a nonnegative integer or a list of nonnegative integers. 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(range(d+1))
sage: H.is_isomorphic(graphs.CompleteGraph(n))
False
sage: H = G.distance_graph(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]
```

TESTS:

Empty input, or unachievable distances silently yield empty graphs.

```
sage: G = graphs.CompleteGraph(5)
sage: G.distance_graph([]).num_edges()
0
sage: G = graphs.CompleteGraph(5)
sage: G.distance_graph(23).num_edges()
0
```

It is an error to provide a distance that is not an integer type.

```
sage: G = graphs.CompleteGraph(5)
sage: G.distance_graph('junk')
Traceback (most recent call last):
...
TypeError: unable to convert x (=junk) to an integer
```

It is an error to provide a negative distance.

```
sage: G = graphs.CompleteGraph(5)
sage: G.distance_graph(-3)
Traceback (most recent call last):
...
ValueError: Distance graph for a negative distance (d=-3) is not defined
```

AUTHOR:

Rob Beezer, 2009-11-25

distance_matrix()

Returns the distance matrix of the (strongly) connected (di)graph.

The distance matrix of a (strongly) connected (di)graph is a matrix whose rows and columns are indexed with the vertices of the (di) graph. The intersection of a row and column contains the respective distance between the vertices indexed at these position.

Warning: The ordering of vertices in the matrix has no reason to correspond to the order of vertices in `vertices()`. In particular, if two integers i, j are vertices of a graph G with distance matrix `M`, then `M[i][i]` is not necessarily the distance between vertices i and j.

EXAMPLES:

```
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*)

Returns 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 (*independent=False*, *value_only=False*, *solver=None*, *verbose=0*)

Returns a minimum dominating set of the graph represented by the list of its vertices. For more information, see the Wikipedia article on dominating sets.

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.

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`). If `independent=True`, computes a minimum independent dominating set.
- `value_only` – boolean (default: `False`)
  - If `True`, only the cardinality of a minimum dominating set is returned.
  - If `False` (default), a minimum dominating set is returned as the list of its 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.

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
```

eccentricity (*v=None*, *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.

INPUT:

- `v` - either a single vertex or a list of vertices. If it is not specified, then it is taken to be all vertices.
- `dist_dict` - optional, a dict of dicts of distance.
- `with_labels` - Whether to return a list or a dict.

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])
[3, 4, 2]
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}
```

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

Returns 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` - if `False`, each edge is a tuple (u, v) of vertices.

EXAMPLES:

```
sage: K = graphs.CompleteBipartiteGraph(9,3)
sage: len(K.edge_boundary( [0,1,2,3,4,5,6,7,8], [9,10,11] ))
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])
[(0, 1, None), (0, 2, None)]
sage: D.edge_boundary([0], labels=False)
[(0, 1), (0, 2)]
```

TESTS:

```
sage: G = graphs.DiamondGraph()
sage: G.edge_boundary([0,1])
[(0, 2, None), (1, 2, None), (1, 3, None)]
sage: G.edge_boundary([0], [0])
[]
```

```
sage: G.edge_boundary([2], [0])
[(0, 2, None)]
```

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

Returns the edge connectivity of the graph. For more information, see the Wikipedia article on connectivity.

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:

- `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 edge cut are returned.
- `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.)
  - 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` – (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:

A basic application on the PappusGraph:

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

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

```
sage: g = graphs.CompleteGraph(5)
sage: [ value, edges, [ setA, setB ]] = g.edge_connectivity(vertices=True)
sage: print 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()) >= g.edge_connectivity()
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: g = graphs.RandomGNP(15,.5)
sage: tree = Graph()
sage: tree.add_edges(g.min_spanning_tree())
sage: for u,v in tree.edge_iterator(labels=None):
...       tree.set_edge_label(u,v,random())
sage: minimum = min([l for u,v,l in tree.edge_iterator()])
sage: [value, [(u,v,l)]] = tree.edge_connectivity(value_only=False, use_edge_labels=True)
sage: l == minimum
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: g.edge_connectivity()
0.0
```

Or if they are just 1-connected

```
sage: g = graphs.PathGraph(10)
sage: g.edge_connectivity()
1.0
```

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

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

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

Returns a minimum edge cut between vertices s and t represented by a list of edges.

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 self is disconnected. For more information, see the Wikipedia article on cuts.

INPUT:

- `s` – source vertex
- `t` – sink vertex
- `value_only` – boolean (default: `True`). When set to `True`, only the weight of a minimum cut is returned. Otherwise, a list of edges of a minimum cut is also returned.
- `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). Otherwise, each edge has weight 1.

- `vertices` – boolean (default: `False`). When set to `True`, returns 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`.

- `method` – There are currently two different implementations of this method :

  - If `method = "FF"` (default), a Python implementation of the Ford-Fulkerson algorithm is used.

  - If `method = "LP"`, the flow problem is solved using 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. 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([l for u,v,l in g.edge_iterator()])
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)))
True
sage: len(set1) + len(set2) == g.order()
True
```

TESTS:

If method is set to an exotic value:

```
sage: g = graphs.PetersenGraph()
sage: g.edge_cut(0,1, method="Divination")
Traceback (most recent call last):
...
ValueError: The method argument has to be equal to either "FF" or "LP"
```

Same result for both methods:

```
sage: g = graphs.RandomGNP(20,.3)
sage: for u,v in g.edges(labels=False):
...       g.set_edge_label(u,v,round(random(),5))
sage: g.edge_cut(0,1, method="FF") == g.edge_cut(0,1,method="LP")
True
```

Rounded return value when using the LP method:

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

edge_disjoint_paths(*s*, *t*, *method='FF'*)

Returns a list of edge-disjoint paths between two vertices as given by Menger's theorem.

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:

- `method` – There are currently two different implementations of this method :
  - If `method = "FF"` (default), a Python implementation of the Ford-Fulkerson algorithm is used.
  - If `method = "LP"`, the flow problem is solved using Linear Programming.

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

EXAMPLE:

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*)

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

INPUT:

- `k` (integer) – the required number of edge-disjoint spanning trees/arborescences
- `root` (vertex) – root of the disjoint arborescences when the graph is directed. If set to `None`, the first vertex in the graph is picked.

- `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:

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

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

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
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 = floor(Integer(g.edge_connectivity())/2)
sage: trees = g.edge_disjoint_spanning_trees(k)
sage: all([t.is_tree() for t in trees])
True
```

REFERENCES:

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

Returns 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` - (default: None) a vertex, a list of vertices or None
- `labels` - if False, each edge is a tuple (u,v) of vertices.
- `ignore_direction` - bool (default: 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=None*)

Returns the label of an edge. Note that if the graph allows multiple edges, then a list of labels on the edge is returned.

EXAMPLES:

```
sage: G = Graph({0 : {1 : 'edgelabel'}}, sparse=True)
sage: G.edges(labels=False)
[(0, 1)]
sage: G.edge_label( 0, 1 )
'edgelabel'
sage: D = DiGraph({0 : {1 : 'edgelabel'}}, sparse=True)
sage: D.edges(labels=False)
[(0, 1)]
sage: D.edge_label( 0, 1 )
'edgelabel'

sage: G = Graph(multiedges=True, sparse=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]
```

TESTS:

```
sage: G = Graph()
sage: G.add_edge(0,1,[7])
sage: G.add_edge(0,2,[7])
sage: G.edge_label(0,1)[0] += 1
sage: G.edges()
[(0, 1, [8]), (0, 2, [7])]
```

edge_labels()
Returns a list of edge labels.

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

edges(*labels=True*, *sort=True*, *key=None*)
Return a list 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` - default: `True` - if `False`, each edge is simply a pair (u,v) of vertices.
- `sort` - default: `True` - if `True`, edges are sorted according to the default ordering.
- `key` - default: `None` - a function 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 list of tuples. It is safe to change the returned list.

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

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),

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
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), (
```

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():
...       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')]
```

TESTS:

It is an error to turn off sorting while providing a key function for sorting.

```
sage: P=graphs.PetersenGraph()
sage: P.edges(sort=False, key=lambda x: x)
Traceback (most recent call last):
...
ValueError: sort keyword is False, yet a key function is given
```

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

Returns 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, 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` - bool (default: True) - if False, each edge is a tuple (u,v) of vertices.
- `sort` - bool (default: True) - 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])
[]
```

TESTS:

```
sage: G = Graph({0:[0]}, loops=True)  # ticket 9581
sage: G.edges_incident(0)
[(0, 0, None)]
```

eigenspaces (*laplacian=False*)

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

INPUT:

- `laplacian` - 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 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 polynomi
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 polynomi
User basis matrix:
[      1      a1 -a1 - 1])
]
```

eigenvectors(*laplacian=False*)

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

INPUT:

- laplacian - 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, [
(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, -
```

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.
```

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

Return a list of edges forming an eulerian circuit if one exists. Otherwise return False.

This is implemented using Hierholzer's algorithm.

INPUT:

- `return_vertices` – (default: `False`) optionally provide a list of vertices for the path
- `labels` – (default: `True`) whether to return edges with labels (3-tuples)
- `path` – (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)]
```

TESTS:

```
sage: Graph({'H': ['G','L','L','D'], 'L': ['G','D']}).eulerian_circuit(labels=False)
[('H', 'D'), ('D', 'L'), ('L', 'G'), ('G', 'H'), ('H', 'L'), ('L', 'H')]
sage: Graph({0: [0, 1, 1, 1]}).eulerian_circuit(labels=False)
[(0, 1), (1, 0), (0, 1), (1, 0), (0, 0)]
```

eulerian_orientation()

Returns 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]
```

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` - 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()`

EXAMPLES:

```
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)],
 [(2, 3), (3, 0), (0, 2)],
 [(0, 3), (3, 1), (1, 0)]]
```

With no embedding provided:

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

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.
```

TESTS:

trac ticket #15551 deprecated the `trace_faces` name:

```
sage: T.trace_faces({0: [1, 3, 2], 1: [0, 2, 3], 2: [0, 3, 1], 3: [0, 1, 2]})
doctest:...: DeprecationWarning: trace_faces is deprecated. Please use faces instead.
See http://trac.sagemath.org/15551 for details.
[[(0, 1), (1, 2), (2, 0)], [(3, 2), (2, 1), (1, 3)], [(2, 3), (3, 0), (0, 2)], [(0, 3), (3,
```

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

Computes 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 Wikipedia article Feedback_vertex_set.

INPUT:

- `value_only` – boolean (default: `False`)
  - When set to `True`, only the minimum cardinal of a minimum vertex set is returned.
  - When set to `False`, the `Set` of vertices of a minimal feedback vertex set 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.
- `constraint_generation` (boolean) – whether to use constraint generation when solving the Mixed Integer Linear Program (default: `True`).

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()
sage: len(feedback)
3
sage: (u,v,l) = cycle.edge_iterator().next()
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
```

TESTS:

Comparing with/without constraint generation:

```
sage: g = digraphs.RandomDirectedGNP(10,.3)
sage: x = g.feedback_vertex_set(value_only = True)
sage: y = g.feedback_vertex_set(value_only = True,
....:           constraint_generation = False)
sage: x == y
True
```

Bad algorithm:

```
sage: g = graphs.PetersenGraph()
sage: g.feedback_vertex_set(constraint_generation = False)
Traceback (most recent call last):
...
ValueError: The only implementation available for undirected graphs is with constraint_gener
```

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

Returns a maximum flow in the graph from `x` to `y` represented by an optimal valuation of the edges. For more information, see the Wikipedia article on maximum 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`)
  - When set to `True`, only the value of a maximal flow is returned.
  - When set to `False`, is returned a pair whose first element is the value of the maximum flow, and whose second value is 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: `False`)
  - When set to `True`, computes an optimal solution under the constraint that the flow going through an edge has to be an integer.
- `use_edge_labels` – boolean (default: `True`)
  - When set to `True`, computes a maximum flow where each edge has a capacity defined by its label. (If an edge has no label, 1 is assumed.)
  - When set to `False`, each edge has capacity 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).
- `method` – There are currently two different implementations of this method :
  - If `method = "FF"`, a Python implementation of the Ford-Fulkerson algorithm is used (only available when `vertex_bound = False`)
  - If `method = "LP"`, the flow problem is solved using Linear Programming.
  - If `method = None` (default), the Ford-Fulkerson implementation is used iif `vertex_bound = False`.
- `solver` – Specify a Linear Program solver to be used. If set to `None`, the default one is used. function of `MixedIntegerLinearProgram`. See the documentation of `MixedIntegerLinearProgram.solve` for more information.

  Only useful when LP is used to solve the flow problem.
- `verbose` (integer) – 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 two 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: g.flow(1,2)
3

sage: b=digraphs.ButterflyGraph(2)
sage: 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: len(flow_graph.edges())
4
```

TESTS:

An exception if raised when forcing "FF" with `vertex_bound = True`:

```
sage: g = graphs.PetersenGraph()
sage: g.flow(0,1,vertex_bound = True, method = "FF")
Traceback (most recent call last):
...
ValueError: This method does not support both vertex_bound=True and method="FF".
```

Or if the method is different from the expected values:

```
sage: g.flow(0,1, method="Divination")
Traceback (most recent call last):
...
ValueError: The method argument has to be equal to either "FF", "LP" or None
```

The two methods are indeed returning the same results (possibly with some numerical noise, cf. trac ticket #12362):

```
sage: g = graphs.RandomGNP(20,.3)
sage: for u,v in g.edges(labels=False):
...       g.set_edge_label(u,v,round(random(),5))
sage: flow_ff = g.flow(0,1, method="FF")
sage: flow_lp = g.flow(0,1,method="LP")
sage: abs(flow_ff-flow_lp) < 0.01
True
```

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

Returns 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.

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

INPUT:

- `set_embedding (boolean)` - whether or not to store an embedding attribute of the computed (minimal) genus of the graph. (Default is True).
- `on_embedding (dict)` - a combinatorial embedding to compute the genus of the graph on. 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. on_embedding takes precedence over a stored _embedding attribute if minimal is set to False. Note that as a shortcut, the user can enter on_embedding=True to compute the genus on the current _embedding attribute. (see eg's.)
- `minimal (boolean)` - 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)` - 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 (boolean)` - whether or not to compute the genus preserving a planar embedding of the boundary. (Default is False). If circular is True, on_embedding is not a valid option.
- `ordered (boolean)` - 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
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 combinatcrial embeddi
3
sage: (graphs.CubeGraph(3)).genus()
0
sage: K23 = graphs.CompleteBipartiteGraph(2,3)
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.set_boundary(['01','10'])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: cube.genus()
0
sage: cube.is_circular_planar()
True
sage: cube.genus(circular=True)
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
```

```
0
sage: cube.genus(circular=True, on_embedding=True)
0
sage: cube.genus(circular=True, 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: Can't 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: Can't 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: K5 = graphs.CompleteGraph(5)
sage: G = K5.copy()
sage: s = 4
sage: for i in range(1,100):
...       k = K5.relabel(range(s,s+5),inplace=False)
...       G.add_edges(k.edges())
...       s += 4
...
sage: G.genus()
100
```

get_boundary()

Returns the boundary of the (di)graph.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.set_boundary([0,1,2,3,4])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: G.get_boundary()
[0, 1, 2, 3, 4]
```

get_embedding()

Returns 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,
```

get_pos(*dim=2*)

Returns the position dictionary, a dictionary specifying the coordinates of each vertex.

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: (...e-17, 1.0),
 ...
 9: (0.475..., 0.154...)}
```

get_vertex(*vertex*)

Retrieve the object associated with a given vertex.

INPUT:

- `vertex` - the given vertex

EXAMPLES:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), 2 : graphs.MoebiusKanto
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.MoebiusKanto
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()

Computes the girth of the graph. For directed graphs, computes the girth of the undirected graph.

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

EXAMPLES:

```
sage: graphs.TetrahedralGraph().girth()
3
sage: graphs.CubeGraph(3).girth()
4
sage: graphs.PetersenGraph().girth()
5
sage: graphs.HeawoodGraph().girth()
6
sage: graphs.trees(9).next().girth()
+Infinity
```

See Also:

- odd_girth() – computes the odd girth of a graph.

TESTS:

Prior to trac ticket #12243, the girth computation assumed vertices were integers (and failed). The example below tests the computation for graphs with vertices that are not integers. In this example the vertices are sets.

```
sage: G = graphs.OddGraph(3)
sage: type(G.vertices()[0])
<class 'sage.sets.set.Set_object_enumerated_with_category'>
sage: G.girth()
5
```

Ticket trac ticket #12355:

```
sage: H=Graph([(0, 1), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4), (1, 6), (2, 5), (3, 4
sage: H.girth()
3
```

Girth < 3 (see trac ticket #12355):

```
sage: g = graphs.PetersenGraph()
sage: g.allow_multiple_edges(True)
sage: g.allow_loops(True)
sage: g.girth()
5
sage: g.add_edge(0,0)
sage: g.girth()
1
sage: g.delete_edge(0,0)
sage: g.add_edge(0,1)
sage: g.girth()
2
sage: g.delete_edge(0,1)
sage: g.girth()
5
sage: g = DiGraph(g)
sage: g.girth()
2
```

graphplot(**options)
Returns a GraphPlot object.

EXAMPLES:

Creating a graphplot object uses the same options as graph.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'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: g.set_boundary([0,1])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: GP = g.graphplot(edge_labels=True, color_by_label=True, edge_style='dashed')
sage: GP.plot()
```

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

```
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_vertices(talk=True)
sage: GP.plot()
```

graphviz_string(*edge_color=None*, *vertex_labels=True*, *edge_labels=False*, *labels='string'*, *color_by_label=False*, *edge_colors=None*, *edge_options=()*, **options*)
Returns 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" or "latex" (default: "string"). If labels is string latex command 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.
- `edge_colors` - (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.
- **`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_options` - a function (or tuple thereof) mapping edges to a dictionary of options for this edge.

EXAMPLES:

```
sage: G = Graph({0:{1:None,2:None}, 1:{0:None,2:None}, 2:{0:None,1:None,3:'foo'}, 3:{2:'foo'
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)
digraph {
  node [shape="plaintext"];
  node_7  [label=" ", texlbl="$\frac{2}{3}$"];
  node_5  [label=" ", texlbl="$\frac{1}{3}$"];
  node_6  [label=" ", texlbl="$\frac{1}{2}$"];
  node_9  [label=" ", texlbl="$1$"];
  node_4  [label=" ", texlbl="$\frac{1}{4}$"];
  node_8  [label=" ", texlbl="$\frac{4}{5}$"];
  node_0  [label=" ", texlbl="$-4$"];
  node_10  [label=" ", texlbl="$2$"];
  node_1  [label=" ", texlbl="$-2$"];
  node_3  [label=" ", texlbl="$-\frac{1}{2}$"];
  node_2  [label=" ", texlbl="$-1$"];
```

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

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

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

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

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

```
}
```

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

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

We now test all options:

```
sage: def edge_options((u,v,label)):
...       options = { "color": { f: "red", g: "blue" }[label] }
...       if (u,v) == (1/2, -2): options["label"]       = "coucou"; options["label_style"] =
...       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)
digraph {
  node_7  [label="2/3"];
  node_5  [label="1/3"];
  node_6  [label="1/2"];
  node_9  [label="1"];
  node_4  [label="1/4"];
  node_8  [label="4/5"];
  node_0  [label="-4"];
  node_10  [label="2"];
  node_1  [label="-2"];
  node_3  [label="-1/2"];
  node_2  [label="-1"];

  node_6 -> node_1 [label="coucou", color = "red"];
```

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

TESTS:

The following digraph has tuples as vertices:

```
sage: print digraphs.ButterflyGraph(1).graphviz_string()
digraph {
  node_3  [label="('1', 1)"];
  node_0  [label="('0', 0)"];
  node_2  [label="('1', 0)"];
  node_1  [label="('0', 1)"];

  node_0 -> node_3;
  node_0 -> node_1;
  node_2 -> node_3;
  node_2 -> node_1;
}
```

The following digraph has vertices with newlines in their string representations:

```
sage: m1 = matrix(3,3)
sage: m2 = matrix(3,3, 1)
sage: m1.set_immutable()
sage: m2.set_immutable()
sage: g = DiGraph({ m1: [m2] })
sage: print g.graphviz_string()
digraph {
  node_0  [label="[0 0 0]\n\
[0 0 0]\n\
[0 0 0]"];
  node_1  [label="[1 0 0]\n\
[0 1 0]\n\
[0 0 1]"];

  node_0 -> node_1;
}
```

REFERENCES:

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

Write a representation in the dot 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'
sage: tempfile = os.path.join(SAGE_TMP, 'temp_graphviz')
sage: G.graphviz_to_file_named(tempfile, edge_labels=True)
sage: print open(tempfile).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'*)

Returns 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 `Graph.traveling_salesman_problem` for 'tsp' algorithm and `find_hamiltonian` from `sage.graphs.generic_graph_pyx` for 'backtrack' algorithm.

INPUT:

- `algorithm` - one of 'tsp' or 'backtrack'.

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):
...
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'])
```

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

Returns True if (u, v) is an edge, False otherwise.

INPUT: The following forms are accepted by NetworkX:

- 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()

Returns 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*)

Returns 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
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')]
```

has_vertex(*vertex*)

Return True if vertex is one of the vertices of this graph.

INPUT:

- `vertex` - an integer

OUTPUT:

- `bool` - True or False

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
```

incidence_matrix(*sparse=True*)

Returns the incidence matrix of the (di)graph.

Each row is a vertex, and each column is an edge. Note that in the case of graphs, there is a choice of orientation for each edge.

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: G.incidence_matrix()
[-1 -1 -1  0  0  0  0  0  0  0  0  0]
[ 0  0  1 -1 -1  0  0  0  0  0  0  0]
```

```
[ 0  1  0  0  0 -1 -1  0  0  0  0  0]
[ 0  0  0  0  1  0  1 -1  0  0  0  0]
[ 1  0  0  0  0  0  0  0 -1 -1  0  0]
[ 0  0  0  1  0  0  0  0  0  1 -1  0]
[ 0  0  0  0  0  1  0  0  1  0  0 -1]
[ 0  0  0  0  0  0  0  1  0  0  1  1]
```

A well known result states that the product of the incidence matrix with its transpose is in fact the Kirchhoff matrix:

```
sage: G = graphs.PetersenGraph()
sage: G.incidence_matrix()*G.incidence_matrix().transpose() == G.kirchhoff_matrix()
True

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

interior_paths(*start*, *end*)

Returns an exhaustive list of paths (also lists) through only interior vertices from vertex start to vertex end in the (di)graph.

Note - start and end do not necessarily have to be boundary vertices.

INPUT:

- `start` - the vertex of the graph to search for paths from
- `end` - the vertex of the graph to search for paths to

EXAMPLES:

```
sage: eg1 = Graph({0:[1,2], 1:[4], 2:[3,4], 4:[5], 5:[6]})
sage: sorted(eg1.all_paths(0,6))
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg2 = copy(eg1)
sage: eg2.set_boundary([0,1,3])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: sorted(eg2.interior_paths(0,6))
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
[[0, 2, 4, 5, 6]]
sage: sorted(eg2.all_paths(0,6))
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg3 = graphs.PetersenGraph()
sage: eg3.set_boundary([0,1,2,3,4])
sage: sorted(eg3.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: sorted(eg3.interior_paths(1,4))
[[1, 6, 8, 5, 7, 9, 4], [1, 6, 9, 4]]
sage: dg = DiGraph({0:[1,3,4], 1:[3], 2:[0,3,4],4:[3]}, boundary=[4])
sage: sorted(dg.all_paths(0,3))
[[0, 1, 3], [0, 3], [0, 4, 3]]
sage: sorted(dg.interior_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, 2, 4, 3], [0, 3], [0, 4, 2, 3], [0, 4, 3]]
sage: sorted(ug.interior_paths(0,3))
[[0, 1, 3], [0, 2, 3], [0, 3]]
```

is_chordal (*certificate=False*, *algorithm='B'*)

Tests 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) [Fulkerson65].

(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) – 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` – Two algorithms are available for this method (see next section), which can be selected by setting `algorithm = "A"` or `algorithm = "B"` (default). 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 $O(m)$ [Rose75] (where m is the number of edges in the graph) but this implementation is not linear because of the complexity of Lex BFS.

Note: Because of a past bug (#11735, #11961), the first implementation (algorithm A) of this method sometimes returned as certificates subgraphs which were **not** holes. Since then, this bug has been fixed and the values are now double-checked before being returned, so that the algorithm only returns correct values or raises an exception. In the case where an exception is raised, the user is advised to switch to the other algorithm. And to **please** report the bug :-)

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 elimintion order:

```
sage: (_, peo) = g.is_chordal(certificate = True)
sage: for v in peo:
...       if not g.subgraph(g.neighbors(v)).is_clique():
...           print "This should never happen !"
...       g.delete_vertex(v)
sage: print "Everything is fine !"
Everything is fine !
```

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
```

TESTS:

This shouldn't raise exceptions (trac ticket #10899):

```
sage: Graph(1).is_chordal()
True
sage: for g in graphs(5):
....:     _ = g.is_chordal()
```

REFERENCES:

TESTS:

Trac Ticket #11735:

```
sage: g = Graph({3:[2,1,4],2:[1],4:[1],5:[2,1,4]})
sage: _, g1 = g.is_chordal(certificate=True); g1.is_chordal()
False
sage: g1.is_isomorphic(graphs.CycleGraph(g1.order()))
True
```

is_circulant(*certificate=False*)

Tests whether the graph is circulant.

For more information on circulant graphs, see the Wikipedia page on circulant graphs.

INPUT:

- `certificate` (boolean) – whether to return a certificate for yes-answers. See OUTPUT section. Set to `False` by default.

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
```

TESTS:

```
sage: digraphs.DeBruijn(3,1).is_circulant(certificate = True)
(True, [(3, [0, 1, 2])])
sage: Graph(1).is_circulant(certificate = True)
(True, (1, []))
sage: Graph(0).is_circulant(certificate = True)
(True, (0, []))
sage: Graph([(0,0)]).is_circulant(certificate = True)
(True, (1, [0]))
```

is_circular_planar(*on_embedding=None*, *kuratowski=False*, *set_embedding=True*, *boundary=None*, *ordered=False*, *set_pos=False*)

Tests 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:

- **kuratowski (boolean) - if set to True, returns a tuple with** boolean first entry and the Kuratowski subgraph or minor as the second entry (see OUTPUT below). It is set to `False` by default.
- **on_embedding (boolean) - the embedding dictionary to test** planarity on. (i.e.: will return `True` or `False` only for the given embedding). It is set to `False` by default.
- **set_embedding (boolean) - 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. It is set to `True` by default.
- `boundary` - a set 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) - whether or not to consider the order of the** boundary. It is set to `False` by default, and required `boundary` to be defined.

- **set_pos - 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_emb 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 or minor 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). For more information, see [Kirkman].

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:

EXAMPLES:

```
sage: g439 = Graph({1:[5,7], 2:[5,6], 3:[6,7], 4:[5,6,7]})
sage: g439.set_boundary([1,2,3,4])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: g439.show()
sage: g439.is_circular_planar(boundary = [1,2,3,4])
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*)

Tests whether a set of vertices is a clique

A clique is a set of vertices such that there is an edge between any two vertices.

INPUT:

- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, defaults to the entire graph.
- `directed_clique` - (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.

EXAMPLES:

```
sage: g = graphs.CompleteGraph(4)
sage: g.is_clique([1,2,3])
True
sage: g.is_clique()
True
sage: h = graphs.CycleGraph(4)
sage: h.is_clique([1,2])
True
sage: h.is_clique([1,2,3])
False
sage: h.is_clique()
False
sage: i = graphs.CompleteGraph(4).to_directed()
sage: i.delete_edge([0,1])
sage: i.is_clique()
True
sage: i.is_clique(directed_clique=True)
False
```

is_connected()

Indicates whether the (di)graph is connected. Note that in a graph, path connected is equivalent to connected.

EXAMPLES:

```
sage: G = Graph( { 0 : [1, 2], 1 : [2], 3 : [4, 5], 4 : [5] } )
sage: G.is_connected()
False
sage: G.add_edge(0,3)
sage: G.is_connected()
True
sage: D = DiGraph( { 0 : [1, 2], 1 : [2], 3 : [4, 5], 4 : [5] } )
sage: D.is_connected()
False
sage: D.add_edge(0,3)
sage: D.is_connected()
True
sage: D = DiGraph({1:[0], 2:[0]})
sage: D.is_connected()
True
```

is_cut_edge(*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

- G.is_cut_edge(1, 2)

•G.is_cut_edge((1, 2))

•G.is_cut_edge(1, 2, 'label')

•G.is_cut_edge((1, 2, 'label'))

OUTPUT:

•Returns True if (u,v) is a cut edge, False otherwise

EXAMPLES:

```
sage: G = graphs.CompleteGraph(4)
sage: G.is_cut_edge(0,2)
False

sage: G = graphs.CompleteGraph(4)
sage: G.add_edge((0,5,'silly'))
sage: G.is_cut_edge((0,5,'silly'))
True

sage: G = Graph([[0,1],[0,2],[3,4],[4,5],[3,5]])
sage: G.is_cut_edge((0,1))
True

sage: G = Graph([[0,1],[0,2],[1,1]])
sage: G.allow_loops(True)
sage: G.is_cut_edge((1,1))
False

sage: G = digraphs.Circuit(5)
sage: G.is_cut_edge((0,1))
False

sage: G = graphs.CompleteGraph(6)
sage: G.is_cut_edge((0,7))
Traceback (most recent call last):
...
ValueError: edge not in graph
```

is_cut_vertex(*u*, *weak=False*)

Returns True if 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:

•`u` – a vertex

•`weak` – (default: `False`) boolean set to $True$ if the connectivity of directed graphs is to be taken in the weak sense, that is ignoring edges orientations.

OUTPUT:

Returns 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: G = graphs.LollipopGraph(4,2)
sage: G.is_cut_vertex(0)
```

```
False
sage: G.is_cut_vertex(3)
True
```

Comparing the weak and strong connectivity of a digraph:

```
sage: D = digraphs.Circuit(6)
sage: D.is_strongly_connected()
True
sage: D.is_cut_vertex(2)
True
sage: D.is_cut_vertex(2, weak=True)
False
```

Giving a vertex that is not in the graph:

```
sage: G = graphs.CompleteGraph(6)
sage: G.is_cut_vertex(7)
Traceback (most recent call last):
...
ValueError: The input vertex is not in the vertex set.
```

is_drawn_free_of_edge_crossings()

Returns True is 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: D.set_planar_positions()
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]])
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)]]

sage: ss = (graphs.WheelGraph(5)).line_graph(labels=False)
sage: ss.is_equitable(prt)
False
```

is_eulerian(*path=False*)

Return true if the graph has a (closed) tour that visits each edge exactly once.

INPUT:

- `path` – 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)
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
```

TESTS:

```
sage: g = Graph({0:[], 1:[], 2:[], 3:[]}); g.is_eulerian()
True
```

is_gallai_tree()

Returns 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 [erdos1978choos].

REFERENCES:

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()

Tests 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 `Graph.traveling_salesman_problem`.

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
```

TESTS:

When no solver is installed, a `OptionalPackageNotFoundError` exception is raised:

```
sage: from sage.misc.exceptions import OptionalPackageNotFoundError
sage: try:
...       g = graphs.ChvatalGraph()
...       if not g.is_hamiltonian():
...          print "There is something wrong here !"
... except OptionalPackageNotFoundError:
...       pass
```

trac ticket #16210:

```
sage: g=graphs.CycleGraph(10)
sage: g.allow_loops(True)
sage: g.add_edge(0,0)
sage: g.is_hamiltonian()
True
```

is_independent_set(*vertices=None*)

Returns True if the set `vertices` is an independent set, False if not. An independent set is a set of vertices such that there is no edge between any two vertices.

INPUT:

- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, defaults to the entire graph.

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 self is an interval graph

INPUT:

- `certificate` (boolean) – The function returns `True` or `False` according to the graph, when `certificate = False` (default). When `certificate = True` and the graph is an interval graph, a dictionary whose keys are the vertices and values are pairs of integers are returned instead of `True`. 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:

A Petersen Graph is not chordal, nor car it be an interval graph

```
sage: g = graphs.PetersenGraph()
sage: g.is_interval()
False
```

Though we can build intervals from the corresponding random generator:

```
sage: g = graphs.RandomIntervalGraph(20)
sage: g.is_interval()
True
```

This method can also return, given an interval graph, a possible embedding (we can actually compute all of them through the PQ-Tree structures):

```
sage: g = Graph(':S__@_@A_@AB_@AC_@ACD_@ACDE_ACDEF_ACDEFG_ACDEGH_ACDEGHI_ACDEGHIJ_ACDEGIJK_A
sage: d = g.is_interval(certificate = True)
sage: print d                                          # not tested
{0: (0, 20), 1: (1, 9), 2: (2, 36), 3: (3, 5), 4: (4, 38), 5: (6, 21), 6: (7, 27), 7: (8, 12
```

From this embedding, we can clearly build an interval graph isomorphic to the previous one:

```
sage: g2 = graphs.IntervalGraph(d.values())
sage: g2.is_isomorphic(g)
True
```

See Also:

- Interval Graph Recognition.
- PQ – Implementation of PQ-Trees.

is_isomorphic(*other*, *certify=False*, *verbosity=0*, *edge_labels=False*)

Tests for isomorphism between self and other.

INPUT:

- certify - 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.

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, certify=True); a
True
sage: from sage.plot.graphics import GraphicsArray
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: GraphicsArray([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, certify=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, certify=True)
(True, {0: 1, 1: 2, 2: 3, 3: 4, 4: 0})
```

TESTS:

```
sage: g1 = '~?A[~~{ACbCwV_~__OOcCW_fAA{CF{CCAAAC__bCCCwOOV___~____OOOOcCCCW___fAAAA'+
sage: g2 = '~?A[??osR?WARSETCJ_QWASehOXQg`QwChK?qSeFQ_sTIaWIV?XIR?KAC?B?`?COCG?o?O_'+
sage: G1 = Graph(g1)
sage: G2 = Graph(g2)
```

```
sage: G1.is_isomorphic(G2)
True
```

Ensure that isomorphic looped graphs with non-range vertex labels report correctly (trac ticket #10814, fixed by trac ticket #8395):

```
sage: G1 = Graph([(0,1), (1,1)])
sage: G2 = Graph([(0,2), (2,2)])
sage: G1.is_isomorphic(G2)
True
sage: G = Graph(multiedges = True, loops = True)
sage: H = Graph(multiedges = True, loops = True)
sage: G.add_edges([(0,1,0),(1,0,1),(1,1,2),(0,0,3)])
sage: H.add_edges([(0,1,3),(1,0,2),(1,1,1),(0,0,0)])
sage: G.is_isomorphic(H, certify=True)
(True, {0: 0, 1: 1})
sage: set_random_seed(0)
sage: D = digraphs.RandomDirectedGNP(6, .2)
sage: D.is_isomorphic(D, certify = True)
(True, {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5})
sage: D.is_isomorphic(D,edge_labels=True, certify = True)
(True, {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5})
```

Ensure that trac trac ticket #11620 is fixed:

```
sage: G1 = DiGraph([(0, 0, 'c'), (0, 4, 'b'), (0, 5, 'c'),
...    (0, 5, 't'), (1, 1, 'c'), (1, 3,'c'), (1, 3, 't'), (1, 5, 'b'),
...    (2, 2, 'c'), (2, 3, 'b'), (2, 4, 'c'),(2, 4, 't'), (3, 1, 't'),
...    (3, 2, 'b'), (3, 2, 'c'), (3, 4, 'c'), (4, 0,'b'), (4, 0, 'c'),
...    (4, 2, 't'), (4, 5, 'c'), (5, 0, 't'), (5, 1, 'b'), (5, 1, 'c'),
...    (5, 3, 'c')], loops=True, multiedges=True)
sage: G2 = G1.relabel({0:4, 1:5, 2:3, 3:2, 4:1,5:0}, inplace=False)
sage: G1.canonical_label(edge_labels=True) == G2.canonical_label(edge_labels=True)
True
sage: G1.is_isomorphic(G2,edge_labels=True)
True
```

Ensure that trac ticket #13114 is fixed

```
sage: g = Graph([(0, 0, 0), (0, 2, 0), (1, 1, 0), (1, 2, 0), (1, 2, 1), (2, 2, 0)])
sage: gg = Graph([(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 2, 0), (2, 2, 0), (2, 2, 1)])
sage: g.is_isomorphic(gg)
False
```

Ensure that trac:14777 is fixed

```
sage: g = Graph()
sage: h = Graph()
sage: g.is_isomorphic(h)
True
```

is_planar(*on_embedding=None*, *kuratowski=False*, *set_embedding=False*, *set_pos=False*)

Returns True if the graph is planar, and False otherwise. 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 graphs.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:

- [1] John M. Boyer and Wendy J. Myrvold, On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. Journal of Graph Algorithms and Applications, Vol. 8, No. 3, pp. 241-273, 2004.

INPUT:

- `kuratowski` - returns a tuple with boolean as first entry. If the graph is nonplanar, will return the Kuratowski subgraph or minor as the second tuple entry. If the graph is planar, returns None as the second entry.
- `on_embedding` - the embedding dictionary to test planarity on. (i.e.: will return True or False only for the given embedding.)
- `set_embedding` - 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 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` - 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_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):
...
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*)

Return `True` if this graph is (k-)regular.

INPUT:

- `k` (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: G.degree_iterator().next()
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_subgraph(*other*, *induced=True*)

Tests 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:

- `induced` - boolean (default: `True`) If set to `True` tests whether `self` is an *induced* subgraph of `other` that is if the vertices of `self` are also vertices of `other`, and the edges of `self` are equal to the edges of `other` between the vertices contained in `self`. If set to ``False tests whether `self` is a subgraph of `other` that is if all vertices of `self` are also in `other` and all edges of `self` are also in `other`.

OUTPUT:

boolean – `True` iff `self` is a (possibly induced) subgraph of `other`.

See Also:

If you are interested in the (possibly induced) subgraphs isomorphic to `self` in `other`, you are looking for the following methods:

- `subgraph_search()` – finds a subgraph isomorphic to G inside of a $self$
- `subgraph_search_count()` – Counts the number of such copies.
- `subgraph_search_iterator()` – Iterate over all the copies of G contained in $self$.

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

TESTS:

Raise an error when self and other are of different types:

```
sage: Graph([(0,1)]).is_subgraph( DiGraph([(0,1)]) )
Traceback (most recent call last):
...
ValueError: The input parameter must be a Graph.
sage: DiGraph([(0,1)]).is_subgraph( Graph([(0,1)]) )
Traceback (most recent call last):
...
ValueError: The input parameter must be a DiGraph.
```

is_transitively_reduced()

Tests whether the digraph is transitively reduced.

A digraph is transitively reduced if it is equal to its transitive reduction.

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()
True
sage: R = graphs.RandomGNP(2000, .01)
sage: R.is_vertex_transitive()
False
```

kirchhoff_matrix(*weighted=None*, *indegree=True*, *normalized=False*, ***kwds*)

Returns the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be $D - M$, 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 $D^{-1/2}(D - M)D^{-1/2}$.

(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` – Binary variable :**
  - 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` – Binary variable :**
  - 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`

  (This variable only matters when the graph is a digraph)

- `normalized` – Binary variable :
  - If `True`, the returned matrix is $D^{-1/2}(D - M)D^{-1/2}$, 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 $D - M$ is returned

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: G.set_boundary([2,3])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: M = G.kirchhoff_matrix(weighted=True, boundary_first=True); M
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
[ 5  0 -3 -2]
[ 0  4 -4  0]
[-3 -4  8 -1]
[-2  0 -1  3]
sage: M = G.kirchhoff_matrix(boundary_first=True); M
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True, sparse=False); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
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: Graph({0:[],1:[2]}).laplacian_matrix(normalized=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]
```

kronecker_product (*other*)

Returns 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 Wikipedia article on the 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

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
```

TESTS:

Tensor product of graphs:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.tensor_product(H)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((0, 'b'), (1, 'a')), ((1, 'a'), (2, 'b')), ((1, 'b'), (2, 'a'))]
sage: T.is_isomorphic( H.tensor_product(G) )
True
```

Tensor product of digraphs:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.tensor_product(J)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((1, 'a'), (2, 'b'))]
```

```
sage: T.is_isomorphic( J.tensor_product(I) )
True
```

The tensor product of two DeBruijn digraphs of same diameter is a DeBruijn digraph:

```
sage: B1 = digraphs.DeBruijn(2, 3)
sage: B2 = digraphs.DeBruijn(3, 3)
sage: T = B1.tensor_product( B2 )
sage: T.is_isomorphic( digraphs.DeBruijn( 2*3, 3) )
True
```

laplacian_matrix(*weighted=None*, *indegree=True*, *normalized=False*, ***kwds*)

Returns the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be $D - M$, 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 $D^{-1/2}(D - M)D^{-1/2}$.

(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` – Binary variable :**
  - 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` – Binary variable :**
  - 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`

  (This variable only matters when the graph is a digraph)
- `normalized` – Binary variable :
  - If `True`, the returned matrix is $D^{-1/2}(D - M)D^{-1/2}$, 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 $D - M$ is returned

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: G.set_boundary([2,3])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: M = G.kirchhoff_matrix(weighted=True, boundary_first=True); M
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
[ 5  0 -3 -2]
[ 0  4 -4  0]
[-3 -4  8 -1]
[-2  0 -1  3]
sage: M = G.kirchhoff_matrix(boundary_first=True); M
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True, sparse=False); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
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: Graph({0:[],1:[2]}).laplacian_matrix(normalized=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]
```

latex_options()

Returns 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*)

Returns a layout for the vertices of this graph.

INPUT:

- layout – one of "acyclic", "circular", "ranked", "graphviz", "planar", "spring", or "tree"
- pos – a dictionary of positions or None (the default)
- save_pos – a boolean
- layout options – (see below)

If layout=algorithm is specified, this 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: g.layout()
{('1', 1): [2.50..., -0.545...],
 ('0', 0): [2.22..., 0.832...],
 ('1', 0): [1.12..., -0.830...],
 ('0', 1): [0.833..., 0.543...]}

sage: 1+1
2
sage: x = g.layout(layout = "acyclic_dummy", save_pos = True)
sage: x =  {('1', 1): [41, 18], ('0', 0): [41, 90], ('1', 0): [140, 90], ('0', 1): [141, 18]

{('1', 1): [41, 18], ('0', 0): [41, 90], ('1', 0): [140, 90], ('0', 1): [141, 18]}


sage: g.layout(dim = 3)
{('1', 1): [1.07..., -0.260..., 0.927...],
```

```
 ('0', 0): [2.02..., 0.528..., 0.343...],
 ('1', 0): [0.674..., -0.528..., -0.343...],
 ('0', 1): [1.61..., 0.260..., -0.927...]}
```

Here is the list of all the available layout options:

```
sage: from sage.graphs.graph_plot import layout_options
sage: for key, value in list(sorted(layout_options.iteritems())):
...       print "option", key, ":", value
option by_component : Whether to do the spring layout by connected component -- a boolean.
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 v
option prog : Which graphviz layout program to use -- one of "circo", "dot", "fdp", "neato",
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 a
```

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*, ***options*)

Computes a circular layout for this graph

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: G = graphs.CirculantGraph(7,[1,3])
sage: G.layout_circular()
{0: [6.12...e-17, 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")
```

layout_default(*by_component=True*, ***options*)

Computes a spring layout for this graph

INPUT:

- `iterations` – a positive integer
- `dim` – 2 or 3 (default: 2)

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) #TODO!!!!
sage: g.layout_spring()
{0: [1.28..., -0.943...],
 1: [1.57..., -0.101...],
 2: [1.83..., 0.747...],
 3: [0.531..., -0.757...],
 4: [0.795..., 0.108...],
 5: [1.08..., 0.946...]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout = "spring")
```

layout_extend_randomly(*pos*, *dim=2*)

Extends randomly a partial layout

INPUT:

- `pos`: a dictionary mapping vertices to positions

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: H.layout_extend_randomly({('0',0): (0,0), ('1',1): (1,1)})
{('1', 1): (1, 1),
 ('0', 0): (0, 0),
 ('1', 0): [0.111..., 0.514...],
 ('0', 1): [0.0446..., 0.332...]}
```

layout_graphviz(*dim=2*, *prog='dot'*, ***options*)

Calls `graphviz` to compute a layout of the vertices of this graph.

INPUT:

- `prog` – one of "dot", "neato", "twopi", "circo", or "fdp"

EXAMPLES:

```
sage: g = digraphs.ButterflyGraph(2)
sage: g.layout_graphviz() # optional - dot2tex, graphviz
{('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...],
 ('...', ...): [...,...]}
sage: g.plot(layout = "graphviz") # optional - dot2tex, graphviz
```

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
sage: g.plot(layout = "graphviz", prog = "neato") # optional - dot2tex, graphviz
sage: g.plot(layout = "graphviz", prog = "twopi") # optional - dot2tex, graphviz
sage: g.plot(layout = "graphviz", prog = "fdp")    # optional - dot2tex, graphviz
sage: g = graphs.BalancedTree(5,2)
sage: g.plot(layout = "graphviz", prog = "circo") # optional - dot2tex, graphviz
```

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, ... are in your path. The graphviz suite can be download from http://graphviz.org.
- Download dot2tex-2.8.?.spkg from http://trac.sagemath.org/sage_trac/ticket/7004 and install it with `sage -i dot2tex-2.8.?.spkg`

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*)

Uses Schnyder's algorithm to compute a planar layout for self, raising an error if self is not planar.

INPUT:

- `set_embedding` - if True, sets the combinatorial embedding used (see self.get_embedding())
- `on_embedding` - dict: provide a combinatorial embedding
- `external_face` - ignored
- `test` - if True, perform sanity tests along the way
- `circular` - ignored

EXAMPLES:

```
sage: g = graphs.PathGraph(10)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.BalancedTree(3,4)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.CycleGraph(7)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.CompleteGraph(5)
sage: g.set_planar_positions(test=True,set_embedding=True)
Traceback (most recent call last):
...
ValueError: Complete graph is not a planar graph
```

layout_ranked(*heights=None*, *dim=2*, *spring=False*, ***options*)

Computes a ranked layout for this graph

INPUT:

- heights – a dictionary mapping heights to the list of vertices at this height

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 = dict( (i,[i, i+3]) for i in range(3) ))
{0: [0.668..., 0],
 1: [0.667..., 1],
 2: [0.677..., 2],
 3: [1.34..., 0],
 4: [1.33..., 1],
 5: [1.33..., 2]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout = "ranked", heights = dict( (i,[i, i+7]) for i in range(7) ))
```

layout_spring(*by_component=True*, ***options*)

Computes a spring layout for this graph

INPUT:

- `iterations` – a positive integer
- `dim` – 2 or 3 (default: 2)

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) #TODO!!!!
sage: g.layout_spring()
{0: [1.28..., -0.943...],
 1: [1.57..., -0.101...],
 2: [1.83..., 0.747...],
 3: [0.531..., -0.757...],
 4: [0.795..., 0.108...],
 5: [1.08..., 0.946...]}
sage: g = graphs.LadderGraph(7)
sage: g.plot(layout = "spring")
```

layout_tree(*tree_orientation='down'*, *tree_root=None*, *dim=2*, ***options*)

Computes an ordered tree layout for this graph, which should be a tree (no non-oriented cycles).

INPUT:

- `tree_root` – the root vertex. By default `None`. In this case, a vertex is chosen close to the center of the tree.
- `tree_orientation` – the direction in which the tree is growing, can be 'up', 'down', 'left' or 'right' (default is 'down')

OUTPUT: a dictionary mapping vertices to positions

EXAMPLES:

```
sage: T = graphs.RandomLobster(25, 0.3, 0.3)
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")

sage: G = graphs.RandomTree(80)
sage: G.plot(layout="tree", tree_orientation = "right")
```

TESTS:

```
sage: G = graphs.CycleGraph(3)
sage: G.plot(layout='tree')
Traceback (most recent call last):
...
RuntimeError: Cannot use tree layout on this graph: self.is_tree() returns False.
```

lex_BFS(*reverse=False*, *tree=False*, *initial_vertex=None*)

Performs a Lex BFS on the graph.

A Lex BFS (or Lexicographic Breadth-First Search) is a Breadth First Search used for the recognition of Chordal Graphs. For more information, see the Wikipedia article on Lex-BFS.

INPUT:

- `reverse` (boolean) – whether to return the vertices in discovery order, or the reverse.

  `False` by default.

- `tree` (boolean) – whether to return the discovery directed tree (each vertex being linked to the one that saw it for the first time)

  `False` by default.

- `initial_vertex` – the first vertex to consider.

  `None` by default.

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.

This algorithm runs in time $O(n^2)$ (where n is the number of vertices in the graph), which is not optimal. An optimal algorithm would run in time $O(m)$ (where m is the number of edges in the graph), and require the use of a doubly-linked list which are not available in python and can not really be written efficiently. This could be done in Cython, though.

EXAMPLE:

A Lex BFS is obviously an ordering of the vertices:

```
sage: g = graphs.PetersenGraph()
sage: len(g.lex_BFS()) == g.order()
True
```

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)
[(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
```

TESTS:

There were some problems with the following call in the past (trac 10899) – now it should be fine:

```
sage: Graph(1).lex_BFS(tree=True)
([0], Digraph on 1 vertex)
```

lexicographic_product (*other*)

Returns 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
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: L = D.lexicographic_product(P); L
Graph on 200 vertices
sage: L.plot() # long time
```

TESTS:

Lexicographic product of graphs:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.lexicographic_product(H)
sage: T.edges(labels=None)
[((0, 'a'), (0, 'b')), ((0, 'a'), (1, 'a')), ((0, 'a'), (1, 'b')), ((0, 'b'), (1, 'a')), ((0
sage: T.is_isomorphic( H.lexicographic_product(G) )
False
```

Lexicographic product of digraphs:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.lexicographic_product(J)
sage: T.edges(labels=None)
[((0, 'a'), (0, 'b')), ((0, 'a'), (1, 'a')), ((0, 'a'), (1, 'b')), ((0, 'b'), (1, 'a')), ((0
sage: T.is_isomorphic( J.lexicographic_product(I) )
False
```

line_graph(*labels=True*)

Returns the line graph of the (di)graph.

INPUT:

- `labels` (boolean) – 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.

  This is set to `True` by default.

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)]
```

Tests:

trac ticket #13787:

```
sage: g = graphs.KneserGraph(7,1)
sage: C = graphs.CompleteGraph(7)
sage: C.is_isomorphic(g)
True
sage: C.line_graph().is_isomorphic(g.line_graph())
True
```

longest_path(*s=None*, *t=None*, *use_edge_labels=False*, *algorithm='MILP'*, *solver=None*, *verbose=0*)

Returns a longest path of `self`.

INPUT:

- `s` (vertex) – 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` (vertex) – 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) – whether the labels on the edges are to be considered as weights (a label set to `None` or `{}` being considered as a weight of 1). Set to `False` by default.
- `algorithm` – one of `"MILP"` (default) or `"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` – (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.

Note: The length of a path is assumed to be the number of its edges, or the sum of their labels.

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()
[(0, 1, None), (1, 2, None), (2, 3, None), (3, 4, None), (4, 9, None), (5, 7, None), (5, 8,
```

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.edges(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
```

TESTS:

The argument `algorithm` must be either `'backtrack'` or `'MILP'`:

```
sage: graphs.PetersenGraph().longest_path(algorithm="abc")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'backtrack' or 'MILP'
```

Disconnected graphs not weighted:

```
sage: g1 = graphs.PetersenGraph()
sage: g2 = 2 * g1
sage: lp1 = g1.longest_path()
sage: lp2 = g2.longest_path()
sage: len(lp1) == len(lp2)
True
```

Disconnected graphs weighted:

```
sage: g1 = graphs.PetersenGraph()
sage: for u,v in g.edges(labels=False):
...       g.set_edge_label(u, v, random())
sage: g2 = 2 * g1
```

```
sage: lp1 = g1.longest_path(use_edge_labels=True)
sage: lp2 = g2.longest_path(use_edge_labels=True)
sage: lp1[0] == lp2[0]
True
```

Empty graphs:

```
sage: Graph().longest_path()
Graph on 0 vertices
sage: Graph().longest_path(use_edge_labels=True)
[0, Graph on 0 vertices]
sage: graphs.EmptyGraph().longest_path()
Graph on 0 vertices
sage: graphs.EmptyGraph().longest_path(use_edge_labels=True)
[0, Graph on 0 vertices]
```

Trivial graphs:

```
sage: G = Graph()
sage: G.add_vertex(0)
sage: G.longest_path()
Graph on 0 vertices
sage: G.longest_path(use_edge_labels=True)
[0, Graph on 0 vertices]
sage: graphs.CompleteGraph(1).longest_path()
Graph on 0 vertices
sage: graphs.CompleteGraph(1).longest_path(use_edge_labels=True)
[0, Graph on 0 vertices]
```

Random test for digraphs:

```
sage: for i in range(20):
...       g = digraphs.RandomDirectedGNP(15, 0.3)
...       for u, v in g.edges(labels=False):
...           g.set_edge_label(u, v, random())
...       lp = g.longest_path()
...       if (not lp.is_directed_acyclic() or
...           not max(lp.out_degree()) <= 1 or
...           not max(lp.in_degree()) <= 1 or
...           not lp.is_connected()):
...           print("Error!")
...           print g.edges()
...           break
```

trac ticket #13019:

```
sage: g = graphs.CompleteGraph(5).to_directed()
sage: g.longest_path(s=1,t=2)
Subgraph of (Complete graph): Digraph on 5 vertices
```

trac ticket #14412:

```
sage: l = [(0, 1), (0, 3), (2, 0)]
sage: G = DiGraph(l)
sage: G.longest_path().edges()
[(0, 1, None), (2, 0, None)]
```

loop_edges()

Returns a list of all loops in the graph.

EXAMPLES:

```
sage: G = Graph(4, 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: 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)]
```

loop_vertices()

Returns 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*)

Returns any loops in the (di)graph.

INPUT:

- `new` – deprecated
- `labels` – 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_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()
[]

sage: G = graphs.PetersenGraph()
sage: G.loops()
[]
```

max_cut(*value_only=True*, *use_edge_labels=False*, *vertices=False*, *solver=None*, *verbose=0*)

Returns a maximum edge cut of the graph. For more information, see the Wikipedia article on cuts.

INPUT:

- `value_only` – boolean (default: `True`)
  - When set to `True` (default), only the value is returned.
  - When set to `False`, both the value and a maximum edge cut are returned.
- `use_edge_labels` – boolean (default: `False`)
  - When set to `True`, computes a maximum weighted cut where each edge has a weight defined by its label. (If an edge has no label, 1 is assumed.)
  - 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. This implies `value_only=False`.
- `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.

EXAMPLE:

Quite obviously, the max cut of a bipartite graph is the number of edges, and the two sets of vertices are the 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 set of vertices to be merged

EXAMPLE:

```
sage: g=graphs.CycleGraph(3)
sage: g.merge_vertices([0,1])
sage: g.edges()
[(0, 2, None)]

sage: # 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)]

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

sage: g=graphs.CycleGraph(5)
sage: g.vertices()
[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)]
```

min_spanning_tree(*weight_function=<function <lambda> at 0x7fb315a3b1b8>, algorithm='Kruskal', starting_vertex=None, check=False*)

Returns the edges of a minimum spanning tree.

INPUT:

- `weight_function` – A function that takes an edge and returns a numeric weight. Defaults to assigning each edge a weight of 1.
- `algorithm` – The algorithm to use in computing a minimum spanning tree of `G`. The default is to use Kruskal's algorithm. The following algorithms are supported:
  - `"Kruskal"` – Kruskal's algorithm.
  - `"Prim_fringe"` – a variant of Prim's algorithm. `"Prim_fringe"` ignores the labels on the edges.
  - `"Prim_edge"` – a variant of Prim's algorithm.
  - `NetworkX` – Uses NetworkX's minimum spanning tree implementation.
- `starting_vertex` – The vertex from which to begin the search for a minimum spanning tree.

- `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()`

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: g.min_spanning_tree(weight_function=weight)
[(3, 4, None), (2, 4, None), (1, 4, None), (0, 4, None)]
sage: g = graphs.PetersenGraph()
sage: g.allow_multiple_edges(True)
sage: g.weighted(True)
sage: g.add_edges(g.edges())
sage: g.min_spanning_tree()
[(0, 1, None), (0, 4, None), (0, 5, None), (1, 2, None), (1, 6, None), (2, 3, None), (2, 7,
```

Prim's algorithm:

```
sage: g = graphs.CompleteGraph(5)
sage: g.min_spanning_tree(algorithm='Prim_edge', starting_vertex=2, weight_function=weight)
[(2, 4, None), (3, 4, None), (1, 4, None), (0, 4, None)]
sage: g.min_spanning_tree(algorithm='Prim_fringe', starting_vertex=2, weight_function=weight
[(2, 4), (4, 3), (4, 1), (4, 0)]
```

multicommodity_flow(*terminals*, *integer=True*, *use_edge_labels=False*, *vertex_bound=False*, *solver=None*, *verbose=0*)

Solves 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 page on multicommodity flows.

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, a intensity of 1 is assumed.
- `integer` (boolean) – whether to require an integer multicommodity flow
- `use_edge_labels` (boolean) – whether to consider the label of edges as numerical values representing a capacity. If set to `False`, a capacity of 1 is assumed
- `vertex_bound` (boolean) – 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` – Specify a Linear Program solver to be used. If set to `None`, the default one is used. function of `MixedIntegerLinearProgram`. See the documentation of `MixedIntegerLinearProgram.solve` for more informations.
- `verbose` (integer) – sets the level of verbosity. Set to 0 by default (quiet).

ALGORITHM:

(Mixed Integer) Linear Program, depending on the value of `integer`.

EXAMPLE:

An easy way to obtain a satisfiable multiflow 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 multiflow 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 multiflow problem has no solution
```

multiple_edges(*to_undirected=False*, *labels=True*)

Returns any multiple edges 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
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')]
```

multiway_cut (*vertices*, *value_only=False*, *use_edge_labels=False*, *solver=None*, *verbose=0*)

Returns a minimum edge multiway cut corresponding to the given set of vertices (cf. http://www.d.kth.se/~viggo/wwwcompendium/node92.html) represented by a list of edges.

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 from C from G.

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)
  - –When set to `True`, only the value of a minimum multiway cut is returned.
  - –When set to `False` (default), the list of edges is returned
- **`use_edge_labels` (boolean)**
  - 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)
  - when set to `False` (default), 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. 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*)

Returns or sets the graph's name.

INPUT:

- `new` - if not None, then this becomes the new name of the (di)graph. (if 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()
''
```

Name of an immutable graph trac ticket #15681

```
sage: g = graphs.PetersenGraph()
sage: gi = g.copy(immutable=True)
sage: gi.name()
'Petersen graph'
sage: gi.name("Hey")
Traceback (most recent call last):
...
NotImplementedError: An immutable graph does not change name
```

neighbor_iterator (*vertex*)

Return an iterator over neighbors of vertex.

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: for i in G.neighbor_iterator('010'):
...       print i
```

```
011
000
110
sage: D = G.to_directed()
sage: for i in D.neighbor_iterator('010'):
...    print i
011
000
110

sage: D = DiGraph({0:[1,2], 3:[0]})
sage: list(D.neighbor_iterator(0))
[1, 2, 3]
```

neighbors(*vertex*)

Return a list of neighbors (in and out if directed) of vertex.

G[vertex] also works.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: sorted(P.neighbors(3))
[2, 4, 8]
sage: sorted(P[4])
[0, 3, 9]
```

networkx_graph(*copy=True*)

Creates a new NetworkX graph from the Sage graph.

INPUT:

- `copy` - if False, and the underlying implementation is a NetworkX graph, then the actual object itself is returned.

EXAMPLES:

```
sage: G = graphs.TetrahedralGraph()
sage: N = G.networkx_graph()
sage: type(N)
<class 'networkx.classes.graph.Graph'>

sage: G = graphs.TetrahedralGraph()
sage: G = Graph(G, implementation='networkx')
sage: N = G.networkx_graph()
sage: G._backend._nxg is N
False

sage: G = Graph(graphs.TetrahedralGraph(), implementation='networkx')
sage: N = G.networkx_graph(copy=False)
sage: G._backend._nxg is N
True
```

num_edges()

Returns the number of edges.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

num_verts()

Returns the number of vertices. Note that len(G) returns the number of vertices in G also.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10

sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

number_of_loops()

Returns 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
```

order()

Returns the number of vertices. Note that len(G) returns the number of vertices in G also.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10

sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

periphery()

Returns the set of vertices in the periphery, i.e. whose eccentricity is equal to the diameter of the (di)graph.

In other words, the periphery is the set of vertices achieving the maximum eccentricity.

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]
```

plot(***options*)

Returns a graphics object representing the (di)graph.

INPUT:

- `pos` - an optional positioning dictionary
- `layout` - what 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` - whether to print vertex labels
- `edge_labels` - whether to print edge labels. By default, False, but 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).
- `vertex_size` - size of vertices displayed
- `vertex_shape` - the shape to draw the vertices (Not available for multiedge digraphs.)
- `graph_border` - whether to include a box around the graph
- `vertex_colors` - 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` - 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. if specified, plot will show each cell in a different color. vertex_colors takes precedence.
- `talk` - if true, prints large vertices with white backgrounds so that labels are legible on slides
- `iterations` - 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` - 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 of the tree to be used as the root for the layout="tree" option. If no root is specified, then one is chosen at random. Ignored unless layout='tree'.

- `tree_orientation` - "up" or "down" (default is "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` - save position computed during plotting

Note:

- 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: list(sorted(graphplot_options.iteritems()))
[...]

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,
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(10)[5:]:
...     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():
...     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
sage: for u,v,l in D.edges():
...     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 = {}
```

```
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.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()
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)

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: (6.12..., 1.0...),
 1: (-0.95..., 0.30...),
 2: (-0.58..., -0.80...),
 3: (0.58..., -0.80...),
 4: (0.95..., 0.30...),
 5: (1.53..., 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]})

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]})
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)

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(layout='tree')

sage: t = DiGraph('JCC???@A??GO??CO??GO??')
sage: t.plot(layout='tree', tree_root=0, tree_orientation="up")
sage: D = DiGraph({0:[1,2,3], 2:[1,4], 3:[0]})
sage: D.plot()

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())
sage: D.plot(edge_labels=True, color_by_label={'a':'blue', 'b':'red'}, edge_style='dashed')

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')

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']},
sage: G.show(pos={'a':[0,1],'b':[1,1],'c':[2,0],'d':[1,0],'e':[0,0]})
```

TESTS:

```
sage: G = DiGraph({0:{1:'a', 2:'a'}, 1:{0:'b'}, 2:{0:'c'}})
sage: p = G.plot(edge_labels=True, color_by_label={'a':'yellow', 'b':'purple'}); p
sage: sorted([x.options()['rgbcolor'] for x in p if isinstance(x, sage.plot.arrow.CurveArrow
['black', 'purple', 'yellow', 'yellow']
```

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='jmol'*, ***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` - 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` - 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` - which renderer to use. Options:
  - `'jmol'` - default
  - `'tachyon'`
- `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
```

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):G.vertices()}, bgcol
```

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):G.vertices()}).show() # long time

sage: C = graphs.CubeGraph(4)
sage: C.plot3d(engine='tachyon', edge_colors={(0,1,0):C.edges()}, vertex_colors={(1,1,1):C.v

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
```

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
sage: D.plot3d().show() # long time

sage: P = graphs.PetersenGraph().to_directed()
sage: from sage.plot.colors import rainbow
sage: edges = P.edges()
sage: R = rainbow(len(edges), 'rgbtuple')
sage: edge_colors = {}
sage: for i in range(len(edges)):
...       edge_colors[R[i]] = [edges[i]]
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']},
sage: G.show3d()
Traceback (most recent call last):
...
NotImplementedError: 3D plotting of multiple edges or loops not implemented.
```

TESTS:

```
sage: G = DiGraph({0:{1:'a', 2:'a'}, 1:{0:'b'}, 2:{0:'c'}})
sage: p = G.plot3d(edge_labels=True, color_by_label={'a':'yellow', 'b':'cyan'})
sage: s = p.x3d_str()
```

This 3D plot contains four yellow objects (two cylinders and two cones), two black objects and 2 cyan objects:

```
sage: s.count("Material diffuseColor='1.0 1.0 0.0'")
4
sage: s.count("Material diffuseColor='0.0 0.0 0.0'")
2
sage: s.count("Material diffuseColor='0.0 1.0 1.0'")
2
```

See Also:

- plot()
- graphviz_string()

radius()

Returns 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.

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
```

TEST:

```
sage: g = Graph()
sage: g.radius()
Traceback (most recent call last):
...
ValueError: This method has no meaning on empty graphs.
```

random_edge(***kwds*)

Returns a random edge of self.

INPUT:

- `**kwds` - arguments to be passed down to the `edge_iterator` method.

EXAMPLE:

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_subgraph(*p*, *inplace=False*)

Return a random subgraph that contains each vertex with prob. p.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.random_subgraph(.25)
Subgraph of (Petersen graph): Graph on 4 vertices
```

random_vertex(***kwds*)

Returns a random vertex of self.

INPUT:

- `**kwds` - arguments to be passed down to the `vertex_iterator` method.

EXAMPLE:

The returned value is a vertex of self:

```
sage: g = graphs.PetersenGraph()
sage: v = g.random_vertex()
sage: v in g
True
```

relabel(*perm=None*, *inplace=True*, *return_map=False*, *check_input=True*, *complete_partial_function=True*)

Relabels the vertices of `self`

INPUT:

- `perm` – a function, dictionary, list, 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 !*.

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]`. Similarly, if `perm` is a list or tuple `l` 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 $\{0, 1, ..., n-1\}$.

Note: at this point, only injective relabeling are supported.

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 a tuple:

```
sage: G.relabel((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]
```

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)
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()
[]
```

Relabeling to simpler labels:

```
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]
```

Recovering the relabeling with `return_map`:

```
sage: G = graphs.CubeGraph(3)
sage: expecting = {'000': 0, '001': 1, '010': 2, '011': 3, '100': 4, '101': 5, '110': 6, '11
sage: G.relabel(return_map=True) == expecting
True

sage: G = graphs.PathGraph(3)
sage: G.relabel(lambda i: i+10, return_map=True)
{0: 10, 1: 11, 2: 12}
```

TESTS:

```
sage: P = Graph(graphs.PetersenGraph())
sage: P.delete_edge([0,1])
sage: P.add_edge((4,5))
sage: P.add_edge((2,6))
sage: P.delete_vertices([0,1])
sage: P.relabel()
```

The attributes are properly updated too

```
sage: G = graphs.PathGraph(5)
sage: G.set_vertices({0: 'before', 1: 'delete', 2: 'after'})
sage: G.set_boundary([1,2,3])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: G.delete_vertex(1)
sage: G.relabel()
sage: G.get_vertices()
{0: 'before', 1: 'after', 2: None, 3: None}
sage: G.get_boundary()
[1, 2]
sage: G.get_pos()
{0: (0, 0), 1: (2, 0), 2: (3, 0), 3: (4, 0)}
```

Check that trac ticket #12477 is fixed:

```
sage: g = Graph({1:[2,3]})
sage: rel = {1:'a', 2:'b'}
sage: g.relabel(rel)
sage: g.vertices()
[3, 'a', 'b']
sage: rel
{1: 'a', 2: 'b'}
```

Immutable graphs cannot be relabeled:

```
sage: Graph(graphs.PetersenGraph(), immutable=True).relabel({})
Traceback (most recent call last):
...
ValueError: To relabel an immutable graph use inplace=False
```

trac ticket #16257:

```
sage: G = graphs.PetersenGraph()
sage: G.relabel( [ i+1 for i in range(G.order()) ], inplace=True )
sage: G.relabel( [ i+1 for i in range(G.order()) ], inplace=True )
```

remove_loops(*vertices=None*)

Removes loops on vertices in vertices. If vertices is None, removes all loops.

EXAMPLE

```
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()
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()

Removes all multiple edges, retaining one edge for each.

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_boundary(*boundary*)

Sets the boundary of the (di)graph.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.set_boundary([0,1,2,3,4])
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: G.get_boundary()
[0, 1, 2, 3, 4]
sage: G.set_boundary((1..4))
sage: G.get_boundary()
[1, 2, 3, 4]
```

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:
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:
sage: SD.plot(pos=posn, vertex_size=400, vertex_colors={'#FFFFFF':range(1,19)}, edge_labels=

sage: G = graphs.HeawoodGraph()
sage: for u,v,l in G.edges():
...     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*)

Sets 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],
sage: G.set_embedding({'s': [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2, 4], 4: [0, 9, 3
Traceback (most recent call last):
...
ValueError: embedding is not valid for Petersen graph
```

set_latex_options(***kwds*)

Sets multiple options for rendering a graph with LaTeX.

INPUTS:

- `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)
True
```

This method is deprecated since Sage-4.4.1.alpha2. Please use instead:

sage: g.layout(layout = "planar", save_pos = True) {0: [1, 1], 1: [2, 2], 2: [3, 2], 3: [1, 4], 4: [5, 1], 5: [0, 5], 6: [1, 0]}

set_pos(*pos*, *dim=2*)
Sets the position dictionary, a dictionary specifying the coordinates of each 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, not str
```

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
```

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.MoebiusKanto
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*, *bidirectional=True*)

Returns a list of vertices representing some shortest path from u to v: if there is no path from u to v, the list is empty.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.
- `bidirectional` - if True, the algorithm will expand vertices from u and v at the same time, making two spheres of half the usual radius. This generally doubles the speed (consider the total volume in each case).

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path(4, 9)
[4, 17, 16, 12, 13, 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: [2], 2: [3], 3: [4], 4: [0] })
sage: G.plot(edge_labels=True).show() # long time
sage: G.shortest_path(0, 3)
[0, 4, 3]
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse = True)
sage: G.shortest_path(0, 3, by_weight=True)
[0, 1, 2, 3]
```

shortest_path_all_pairs(*by_weight=False*, *default_weight=1*, *algorithm='auto'*)

Computes a shortest path between each pair of vertices.

INPUT:

- **`by_weight` - Whether to use the labels defined over the edges as** weights. If `False` (default), the distance between u and v is the minimum number of edges of a path from u to v.
- **`default_weight` - (defaults to 1) The default weight to assign** edges that don't have a weight (i.e., a label).

  Implies `by_weight == True`.
- `algorithm` – four options :
  - `"BFS"` – the computation is done through a BFS centered on each vertex successively. Only implemented when `default_weight = 1` and `by_weight = False`.
  - `"Floyd-Warshall-Cython"` – through the Cython implementation of the Floyd-Warshall algorithm.
  - `"Floyd-Warshall-Python"` – through the Python implementation of the Floyd-Warshall algorithm.
  - `"auto"` – use the fastest algorithm depending on the input (`"BFS"` if possible, and `"Floyd-Warshall-Python"` otherwise)

This is the default value.

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: Three different implementations are actually available through this method :

•BFS (Cython)

•Floyd-Warshall (Cython)

•Floyd-Warshall (Python)

The BFS algorithm is the fastest of the three, then comes the Cython implementation of Floyd-Warshall, and last the Python implementation. The first two implementations, however, only compute distances based on the topological distance (each edge is of weight 1, or equivalently the length of a path is its number of edges). Besides, they do not deal with graphs larger than 65536 vertices (which already represents 16GB of ram).

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:

```
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,
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,
sage: pred[0]
{0: None, 1: 0, 2: 1, 3: 2, 4: 0}
```

So for example the shortest weighted path from 0 to 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(by_weight = True)
({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}})
sage: G.shortest_path_all_pairs(default_weight=200)
({0: {0: 0, 1: 200, 2: 5, 3: 4, 4: 2},
1: {0: 200, 1: 0, 2: 200, 3: 201, 4: 202},
2: {0: 5, 1: 200, 2: 0, 3: 1, 4: 3},
3: {0: 4, 1: 201, 2: 1, 3: 0, 4: 2},
4: {0: 2, 1: 202, 2: 3, 3: 2, 4: 0}},
{0: {0: None, 1: 0, 2: 3, 3: 4, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 4, 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}})
```

Checking the 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: d1 == d2 == d3
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
```

TESTS:

Wrong name for `algorithm`:

```
sage: g.shortest_path_all_pairs(algorithm="Bob")
Traceback (most recent call last):
...
ValueError: The algorithm keyword can only be set to "auto", "BFS", "Floyd-Warshall-Python"
```

shortest_path_length(*u*, *v*, *by_weight=False*, *bidirectional=True*, *weight_sum=None*)

Returns the minimal length of paths from u to v.

If there is no path from u to v, returns Infinity.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.
- `bidirectional` - if True, the algorithm will expand vertices from u and v at the same time, making two spheres of half the usual radius. This generally doubles the speed (consider the total volume in each case).

- `weight_sum` - if False, returns the number of edges in the path. If True, returns the sum of the weights of these edges. Default behavior is to have the same value as by_weight.

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path_length(4, 9)
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
```

shortest_path_lengths(*u*, *by_weight=False*, *weight_sums=None*)

Returns a dictionary of shortest path lengths keyed by targets that are connected by a path from u.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.

EXAMPLES:

```
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:
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}
```

shortest_paths(*u*, *by_weight=False*, *cutoff=None*)

Returns a dictionary associating to each vertex v a shortest path from u to v, if it exists.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, uses Dijkstra's algorithm to find the shortest paths by weight.
- `cutoff` - integer depth to stop search.

  (ignored if `by_weight == True`)

EXAMPLES:

```
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
```

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
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], 1
```

```
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]}
```

show(*method='matplotlib'*, ***kwds*)

Shows the (di)graph.

INPUT:

- `method` –
  - If `method="matplotlib"` (default) then graph is drawn as a picture file, then displayed. In this situation, the method accepts any other option understood by `plot()` (graph-specific) or by `sage.plot.graphics.Graphics.show()`.
  - If `method="js"` the graph is displayed using the d3.js library in a browser. In this situation, the method accepts any other option understood by `sage.graphs.graph_plot_js.gen_html_code()`. Depending on whether d3js optional package is installed or not, the javascript code will be used locally or fetched from d3js.org website by the browser.

This method accepts any other option understood by `plot()` (graph-specific) or by `sage.plot.graphics.Graphics.show()`.

Note:

- See the documentation of the `sage.graphs.graph_plot` module for information on default arguments of this method.
- For the javascript counterpart, refer to `sage.graphs.graph_plot_js`.

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='jmol'*, ***kwds*)

Plots the graph using Tachyon, and shows the resulting plot.

INPUT:

- `bgcolor` - rgb tuple (default: (1,1,1))
- `vertex_size` - float (default: 0.06)
- `vertex_colors` - 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` - 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.
- `edge_size` - float (default: 0.02)
- `edge_size2` - float (default: 0.0325), used for Tachyon sleeves
- `pos3d` - a position dictionary for the vertices
- `iterations` - how many iterations of the spring layout algorithm to go through, if applicable

- `engine` - which renderer to use. Options:
- `'jmol'` - default 'tachyon'
- `xres` - resolution
- `yres` - resolution
- `**kwds` - passed on to the Tachyon command

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):G.v
```

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):G.vertices()}) # long time

sage: C = graphs.CubeGraph(4)
sage: C.show3d(engine='tachyon', edge_colors={(0,1,0):C.edges()}, vertex_colors={(1,1,1):C.v

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
```

size()

Returns the number of edges.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

spanning_trees_count(*root_vertex=None*)

Returns 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` – integer (default: the first vertex) This is the vertex that will be used as root for all spanning out-trees if the graph is a directed graph. 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
```

spectrum(*laplacian=False*)

Returns a list of the eigenvalues of the adjacency matrix.

INPUT:

•laplacian - if True, use the Laplacian matrix (see kirchhoff_matrix())

OUTPUT:

A list of the eigenvalues, including multiplicities, sorted with the largest eigenvalue first.

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]
```

TESTS:

The Laplacian matrix of a graph is the negative of the adjacency matrix with the degree of each vertex on the diagonal. So for a regular graph, if δ is an eigenvalue of a regular graph of degree r, then $r - \delta$ will be an eigenvalue of the Laplacian. The Hoffman-Singleton graph is regular of degree 7, so the following will test both the Laplacian construction and the computation of eigenvalues.

```
sage: H = graphs.HoffmanSingletonGraph()
sage: evals = H.spectrum()
sage: lap = map(lambda x : 7 - x, evals)
sage: lap.sort(reverse=True)
sage: lap == H.spectrum(laplacian=True)
True
```

steiner_tree(*vertices*, *weighted=False*, *solver=None*, *verbose=0*)

Returns 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.

Wikipedia article on Steiner Trees.

INPUT:

- `vertices` – the vertices to be connected by the Steiner Tree.
- `weighted` (boolean) – 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` – (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: st = g.steiner_tree(g.vertices()[:5])
sage: st.is_tree()
True
```

And all the 5 vertices are contained in this tree

```
sage: all([v in st for v in g.vertices()[:5] ])
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):
...
EmptySetError: The given vertices do not all belong to the same connected component. This pr
```

strong_product(*other*)

Returns 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

sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: S = D.strong_product(P); S
Graph on 200 vertices
sage: S.plot() # long time
```

TESTS:

Strong product of graphs is commutative:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.strong_product(H)
sage: T.is_isomorphic( H.strong_product(G) )
True
```

Strong product of digraphs is commutative:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.strong_product(J)
sage: T.is_isomorphic( J.strong_product(I) )
True
```

Counting the edges (see trac ticket #13699):

```
sage: g = graphs.RandomGNP(5,.5)
sage: gn,gm = g.order(), g.size()
sage: h = graphs.RandomGNP(5,.5)
sage: hn,hm = h.order(), h.size()
sage: product_size = g.strong_product(h).size()
sage: expected = gm*hn + hm*gn + 2*gm*hm
sage: if product_size != expected:
...       print "Something is really wrong here...", product_size, "!=", expected
```

subdivide_edge(**args*)

Subdivides 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`

EXAMPLE:

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 = g.edges()[0]
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 sorted(g.edges())
[(0, 1, 'label1'), (1, 2, 'label2')]
```

Specifying the label:

```
sage: g.subdivide_edge(0,1,"label1", 3)
sage: print sorted(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 sorted(g.edges())
[(0, 3, 'label1'), (1, 5, 'label1'), (1, 6, 'label2'), (2, 10, 'label2'), (3, 4, 'label1'),
```

If too many arguments are given, an exception is raised

```
sage: g.subdivide_edge(0,1,1,1,1,1,1,1,1,1)
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

TESTS:

trac ticket #15895 is fixed:

```
sage: F = graphs.PathGraph(3)
sage: S = 'S'; F.add_vertex(S)
sage: F.add_edge(S,0)
sage: F2 = Graph(F)
sage: F2.subdivide_edges(list(F2.edges(labels=False)),2)
sage: 0 in F2.degree()
False
```

subdivide_edges (*edges*, *k*)

Subdivides 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.

EXAMPLE:

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 = [P.edges()[0] 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*)

Returns 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` - Using inplace is True will simply delete the extra vertices and edges from the current graph. This will modify the graph.
- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, 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.). If not edges are not specified, then 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` - If specified, this is expected to be a function on vertices, which is intersected with the vertices specified, if any are.
- `edge_property` - If specified, this is expected to be a function on edges, which is intersected with the edges specified, if any are.
- `algorithm` - If `algorithm=delete` or `inplace=True`, then the graph is constructed by deleting edges and vertices. If `add`, then the graph is constructed by building a new graph from the appropriate vertices and edges. If not specified, then the algorithm is chosen based on the number of vertices in the subgraph.

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 = graphs.CompleteGraph(9).to_directed()
sage: H = D.subgraph([0,1,2]); H
Subgraph of (Complete graph): 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 graph: Digraph on 9 vertices
sage: D.subgraph([0,1,2], inplace=True); D
Subgraph of (Complete graph): 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]
sage: G.subgraph(vertices=G.vertices())==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 : v%2 == 0)
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(range(10)) # uses the 'add' algorithm
Subgraph of (Path Graph): Graph on 10 vertices
```

TESTS: The appropriate properties are preserved.

```
sage: g = graphs.PathGraph(10)
sage: g.is_planar(set_embedding=True)
True
sage: g.set_vertices(dict((v, 'v%d'%v) for v in g.vertices()))
sage: h = g.subgraph([3..5])
sage: h.get_pos().keys()
[3, 4, 5]
sage: h.get_vertices()
{3: 'v3', 4: 'v4', 5: 'v5'}
```

subgraph_search(*G*, *induced=False*)

Returns a copy of `G` in `self`.

INPUT:

- `G` – the 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: This method also works on digraphs.

See Also:

- `subgraph_search_count()` – Counts the number of copies of a graph H inside of a graph G
- `subgraph_search_iterator()` – Iterate on the copies of a graph H inside of a graph G

ALGORITHM:

Brute-force search.

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
```

TESTS:

Inside of a small graph (trac ticket #13906):

```
sage: Graph(5).subgraph_search(Graph(1))
Graph on 1 vertex
```

subgraph_search_count(*G*, *induced=False*)

Returns the number of labelled occurences of `G` in `self`.

INPUT:

- G – the graph whose copies we are looking for in self.
- induced – boolean (default: False). Whether or not to count induced copies of G in self.

ALGORITHM:

Brute-force search.

Note: This method also works on digraphs.

See Also:

- subgraph_search() – finds an subgraph isomorphic to H inside of a graph G
- subgraph_search_iterator() – Iterate on 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 = DiGraph()
sage: T5.add_edges([(i,j) for i in xrange(5) for j in xrange(i+1, 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 = T5.subgraph([0,1,2])
sage: T5.subgraph_search_count(T3)
10
sage: binomial(5,3)
10
```

The empty graph is a subgraph of every graph:

```
sage: g.subgraph_search_count(graphs.EmptyGraph())
1
```

TESTS:

Inside of a small graph (trac ticket #13906):

```
sage: Graph(5).subgraph_search_count(Graph(1))
5
```

subgraph_search_iterator(*G*, *induced=False*)

Returns 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`.

ALGORITHM:

Brute-force search.

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 a graph G
- `subgraph_search_count()` – Counts the number of copies of a graph H inside of a graph G

EXAMPLE:

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]
```

TESTS:

Inside of a small graph (trac ticket #13906):

```
sage: list(Graph(5).subgraph_search_iterator(Graph(1)))
[Graph on 1 vertex, Graph on 1 vertex, Graph on 1 vertex, Graph on 1 vertex, Graph on 1 vert
```

szeged_index()

Returns 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 graph is then defined as [1]: $\sum_{uv \in E(G)} n_u(uv) \times n_v(uv)$

EXAMPLE:

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*)

Returns 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 Wikipedia article on the 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

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
```

TESTS:

Tensor product of graphs:

```
sage: G = Graph([(0,1), (1,2)])
sage: H = Graph([('a','b')])
sage: T = G.tensor_product(H)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((0, 'b'), (1, 'a')), ((1, 'a'), (2, 'b')), ((1, 'b'), (2, 'a'))]
sage: T.is_isomorphic( H.tensor_product(G) )
True
```

Tensor product of digraphs:

```
sage: I = DiGraph([(0,1), (1,2)])
sage: J = DiGraph([('a','b')])
sage: T = I.tensor_product(J)
sage: T.edges(labels=None)
[((0, 'a'), (1, 'b')), ((1, 'a'), (2, 'b'))]
sage: T.is_isomorphic( J.tensor_product(I) )
True
```

The tensor product of two DeBruijn digraphs of same diameter is a DeBruijn digraph:

```
sage: B1 = digraphs.DeBruijn(2, 3)
sage: B2 = digraphs.DeBruijn(3, 3)
sage: T = B1.tensor_product( B2 )
sage: T.is_isomorphic( digraphs.DeBruijn( 2*3, 3) )
True
```

to_dictionary(*edge_labels=False*, *multiple_edges=False*)

Returns the graph as a dictionary.

INPUT:

- `edge_labels` (boolean) – whether to include edge labels in the output.
- `multiple_edges` (boolean) – 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 "neigbours" 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]),
 (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: {8: None, 2: None, 4: None},
 4: {0: None, 9: None, 3: None},
 5: {0: None, 8: None, 7: None},
 6: {8: None, 1: None, 9: None},
 7: {9: None, 2: None, 5: 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: {8: [None], 2: [None], 4: [None]},
 4: {0: [None], 9: [None], 3: [None]},
 5: {0: [None], 8: [None], 7: [None]},
 6: {8: [None], 1: [None], 9: [None]},
 7: {9: [None], 2: [None], 5: [None]},
 8: {3: [None], 5: [None], 6: [None]},
 9: {4: [None], 6: [None], 7: [None]}}
```

to_simple()

Returns a simple version of itself (i.e., undirected and loops and multiple edges are removed).

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
```

trace_faces(**args*, ***kwds*)

Deprecated: Use `faces()` instead. See trac ticket #15551 for details.

transitive_closure()

Computes the transitive closure of a graph and returns it. The original graph is not modified.

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 graph is a complete graph. In particular, the transitive closure of a connected undirected graph is a complete graph. The transitive closure of a directed acyclic graph is a directed acyclic graph representing the full partial order.

EXAMPLES:

```
sage: g=graphs.PathGraph(4)
sage: g.transitive_closure()
Transitive closure of Path Graph: Graph on 4 vertices
sage: g.transitive_closure()==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)]
```

transitive_reduction()

Returns a transitive reduction of a graph. The original graph is not modified.

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: edges = g.transitive_reduction().edges(); len(edges)
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*, *solver=None*, *constraint_generation=None*, *verbose=0*, *verbose_constraints=False*)

Solves 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) – whether to consider the weights of the edges.
  - If set to `False` (default), all edges are assumed to weight 1
  - If set to `True`, the weights are taken into account, and the circuit returned is the one minimizing the sum of the weights (an edge with no label is assumed to have weight 1).
- `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:

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.edges(labels = None):
...       g.set_edge_label(u,v,1)

sage: cycle = graphs.CycleGraph(10)
sage: for u,v in cycle.edges(labels = None):
...       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):
...       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
```

TESTS:

Comparing the results returned according to the value of `constraint_generation`. First, for graphs:

```
sage: from operator import itemgetter
sage: n = 20
sage: for i in range(20):
```

```
...       g = Graph()
...       g.allow_multiple_edges(False)
...       for u,v in graphs.RandomGNP(n,.2).edges(labels = False):
...            g.add_edge(u,v,round(random(),5))
...       for u,v in graphs.CycleGraph(n).edges(labels = False):
...            if not g.has_edge(u,v):
...                g.add_edge(u,v,round(random(),5))
...       v1 = g.traveling_salesman_problem(constraint_generation = False, use_edge_labels =
...       v2 = g.traveling_salesman_problem(use_edge_labels = True)
...       c1 = sum(map(itemgetter(2), v1.edges()))
...       c2 = sum(map(itemgetter(2), v2.edges()))
...       if c1 != c2:
...           print "Error !",c1,c2
...           break
```

Then for digraphs:

```
sage: from operator import itemgetter
sage: set_random_seed(0)
sage: n = 20
sage: for i in range(20):
...       g = DiGraph()
...       g.allow_multiple_edges(False)
...       for u,v in digraphs.RandomDirectedGNP(n,.2).edges(labels = False):
...            g.add_edge(u,v,round(random(),5))
...       for u,v in digraphs.Circuit(n).edges(labels = False):
...            if not g.has_edge(u,v):
...                g.add_edge(u,v,round(random(),5))
...       v2 = g.traveling_salesman_problem(use_edge_labels = True)
...       v1 = g.traveling_salesman_problem(constraint_generation = False, use_edge_labels =
...       c1 = sum(map(itemgetter(2), v1.edges()))
...       c2 = sum(map(itemgetter(2), v2.edges()))
...       if c1 != c2:
...           print "Error !",c1,c2
...           print "With constraint generation :",c2
...           print "Without constraint generation :",c1
...           break
```

Simple tests for multiple edges and loops:

```
sage: G = DiGraph(multiedges=True, loops=True)
sage: G.is_hamiltonian()
Traceback (most recent call last):
...
ValueError: The traveling salesman problem (or finding Hamiltonian cycle) is not well define
sage: G.add_vertex(0)
sage: G.is_hamiltonian()
Traceback (most recent call last):
...
ValueError: The traveling salesman problem (or finding Hamiltonian cycle) is not well define
sage: G.add_edge(0,0,1)
sage: G.add_edge(0,0,2)
sage: tsp = G.traveling_salesman_problem(use_edge_labels=True)
Traceback (most recent call last):
...
ValueError: The traveling salesman problem (or finding Hamiltonian cycle) is not well define
sage: G.add_vertex(1)
sage: G.is_hamiltonian()
False
```

```
sage: G.add_edge(0,1,2)
sage: G.add_edge(0,1,3)
sage: G.add_edge(1,1,1)
sage: G.add_edge(1,0,2)
sage: G.is_hamiltonian()
True
sage: tsp = G.traveling_salesman_problem(use_edge_labels=True)
sage: sum(tsp.edge_labels())
4
```

Graphs on 2 vertices:

```
sage: Graph([(0,1),(0,1)],multiedges=True).is_hamiltonian()
True
sage: DiGraph([(0,1),(0,1)],multiedges=True).is_hamiltonian()
False
sage: DiGraph([(0,1),(1,0)],multiedges=True).is_hamiltonian()
True
sage: G = DiGraph(loops=True)
sage: G.add_edges([(0,0),(0,1),(1,1),(1,0)])  # i.e. complete digraph with loops
sage: G.is_hamiltonian()
True
sage: G.remove_loops()
sage: G.is_hamiltonian()
True
sage: G.allow_loops(False)
sage: G.is_hamiltonian()
True
```

Check that weight 0 edges are handled correctly (see trac ticket #16214):

```
sage: G = Graph([(0,1,1),(0,2,0),(0,3,1),(1,2,1),(1,3,0),(2,3,1)])
sage: tsp = G.traveling_salesman_problem(use_edge_labels=True)
sage: sum(tsp.edge_labels())
2
```

triangles_count(*algorithm='iter'*)

Returns the number of triangles in the (di)graph.

For digraphs, we count the number of directed circuit of length 3.

INPUT:

- `algorithm` – (default: `'matrix'`) specifies the algorithm to use among:
  - `'matrix'` uses the trace of the cube of the adjacency matrix.
  - `'iter'` iterates over the pairs of neighbors of each vertex. This is faster for sparse graphs.

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(150)
sage: G.triangles_count() == binomial(150,3)
True
```

The 2-dimensional DeBruijn graph of 2 symbols has 2 directed C3:

```
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
```

TESTS:

Comparison on algorithms:

```
sage: for i in xrange(10): # long test
...       G = graphs.RandomBarabasiAlbert(50,2)
...       tm = G.triangles_count(algorithm='matrix')
...       te = G.triangles_count(algorithm='iter')
...       if tm!=te:
...          print "That's not good!"
```

Asking for an unknown algorithm:

```
sage: G = Graph()
sage: G.triangles_count(algorithm='tip top')
Traceback (most recent call last):
...
ValueError: Algorithm 'tip top' not yet implemented. Please contribute.
```

union(*other*)

Returns 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 immutable, the resulting graph is immutable.

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)]
```

TESTS:

Multiple edges and loops (trac ticket #15627):

```
sage: g = Graph(multiedges=True, loops=True)
sage: g.add_edges(graphs.PetersenGraph().edges())
sage: g.add_edges(graphs.PetersenGraph().edges())
sage: g.add_edge(0,0)
sage: g.add_edge(0,0,"Hey")
sage: g.add_edge(0,9)
sage: g.add_edge(0,9)
sage: g.add_edge(0,9)
sage: (2*g.size()) == (2*g).size()
True
```

Immutable input ? Immutable output (trac ticket #15627):

```
sage: g = g.copy(immutable=True)
sage: (2*g)._backend
<class 'sage.graphs.base.static_sparse_backend.StaticSparseBackend'>
```

vertex_boundary(*vertices1*, *vertices2=None*)

Returns 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', '
sage: G.vertex_boundary(['0000', '1111'], l)
['0111', '0001', '0010', '0100', '1101', '1011']

sage: D = DiGraph({0:[1,2], 3:[0]})
sage: D.vertex_boundary([0])
[1, 2]
```

vertex_connectivity(*value_only=True*, *sets=False*, *solver=None*, *verbose=0*)

Returns the vertex connectivity of the graph. For more information, see the Wikipedia article on connectivity.

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.
- 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 size $k-1$. For this reason, the certificates returned in this situation are empty.

INPUT:

- `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)
  – When set to True, also returns the two sets of vertices that are disconnected by the cut. Implies value_only=False
- 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:

A basic application on a PappusGraph:

```
sage: g=graphs.PappusGraph()
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 ]] = g.vertex_connectivity(sets=True)
sage: len(setA) == 1 or len(setB) == 1
True
```

A vertex cut in a tree is any internal vertex:

```
sage: g = graphs.RandomGNP(15,.5)
sage: tree = Graph()
sage: tree.add_edges(g.min_spanning_tree())
sage: [val, [cut_vertex]] = tree.vertex_connectivity(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: g.vertex_connectivity()
0
```

Or if they are just 1-connected:

```
sage: g = graphs.PathGraph(10)
sage: g.vertex_connectivity()
1
```

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

```
sage: g = digraphs.ButterflyGraph(3)
sage: g.vertex_connectivity()
0
```

A complete graph on 10 vertices is 9-connected:

```
sage: g = graphs.CompleteGraph(10)
sage: g.vertex_connectivity()
9
```

A complete digraph on 10 vertices is 9-connected:

```
sage: g = DiGraph(graphs.CompleteGraph(10))
sage: g.vertex_connectivity()
9
```

vertex_cut(*s*, *t*, *value_only=True*, *vertices=False*, *solver=None*, *verbose=0*)

Returns 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 on cuts.

INPUT:

- `value_only` – boolean (default: `True`). When set to `True`, only the size of the minimum cut is returned.
- `vertices` – boolean (default: `False`). When set to `True`, also returns the two sets of vertices that are disconnected by the cut. Implies `value_only` set to False.
- `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:

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

EXAMPLE:

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 == 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
sage: len(set2) == 1
True
```

vertex_disjoint_paths(*s*, *t*)

Returns a list of vertex-disjoint paths between two vertices as given by Menger's theorem.

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.

EXAMPLE:

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*)

Returns 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
sage: timeit V = list(P._nxg.adj.iterkeys())     # not tested
100000 loops, best of 3: 3.45 [micro]s per loop
```

In other words, if you want a fast vertex iterator, call the dictionary directly.

vertices(*key=None*, *boundary_first=False*)

Return a list of the vertices.

INPUT:

- `key` - 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.
- `boundary_first` - default: `False` - if `True`, return the boundary vertices first.

OUTPUT:

The vertices of the list.

Warning: There is always an attempt to sort the list before returning the result. However, 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 sort, 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 alternatives. The moral is: if you want a fast vertex iterator, call the dictionary directly.

```
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
sage: timeit V = list(P._nxg.adj.iterkeys())     # not tested
100000 loops, best of 3: 3.45 [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)]
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.discri
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]
```

If boundary vertices are requested first, then they are sorted separately from the remainder (which are also sorted).

```
sage: P = graphs.PetersenGraph()
sage: P.set_boundary((5..9))
doctest:...: DeprecationWarning: The boundary parameter is deprecated and will soon disappea
See http://trac.sagemath.org/15494 for details.
sage: P.vertices(boundary_first=True)
[5, 6, 7, 8, 9, 0, 1, 2, 3, 4]
```

```
sage: P.vertices(boundary_first=True, key=lambda x: -x)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

weighted(*new=None*)

Whether the (di)graph is to be considered as a weighted (di)graph.

INPUT:

- `new` (optional bool): 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
```

TESTS:

Ensure that trac ticket #10490 is fixed: allows a weighted graph to be set as unweighted.

```
sage: G = Graph({1:{2:3}})
sage: G.weighted()
False
sage: G.weighted('a')
sage: G.weighted(True)
sage: G.weighted()
True
sage: G.weighted('a')
sage: G.weighted()
True
sage: G.weighted(False)
sage: G.weighted()
False
sage: G.weighted('a')
sage: G.weighted()
False
```

```
sage: G.weighted(True)
sage: G.weighted()
True
```

Ensure that graphs using the static sparse backend can not be mutated using this method, as fixed in trac ticket #15278:

```
sage: G = graphs.PetersenGraph()
sage: G.weighted()
False
sage: H = copy(G)
sage: H == G
True
sage: H.weighted(True)
sage: H == G
False
sage: G_imm = Graph(G, data_structure="static_sparse")
sage: G_imm == G
True
sage: G_imm.weighted()
False
sage: G_imm.weighted(True)
Traceback (most recent call last):
...
TypeError: This graph is immutable and can thus not be changed.
Create a mutable copy, e.g., by `g.copy(immutable=False)`
sage: G_mut = G_imm.copy(immutable=False)
sage: G_mut == G_imm
True
sage: G_mut.weighted(True)
sage: G_mut == G_imm
False
sage: G_mut == H
True
```

weighted_adjacency_matrix(*sparse=True*, *boundary_first=False*)

Returns the weighted adjacency matrix of the graph.

Each vertex is represented by its position in the list returned by the vertices() function.

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
```

The following doctest verifies that #4888 is fixed:

```
sage: G = DiGraph({0:{}, 1:{0:1}, 2:{0:1}}, weighted = True,sparse=True)
sage: G.weighted_adjacency_matrix()
[0 0 0]
[1 0 0]
[1 0 0]
```

wiener_index(*G*)

Returns the Wiener index of the graph.

The Wiener index of a graph G can be defined in two equivalent ways [KRG96b] :

- $W(G) = \frac{1}{2}\sum_{u,v\in G} d(u,v)$ 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 $W(G) = \sum_{e\in E(G)} |\Omega(e)|$.

EXAMPLE:

From [GYLL93c], cited in [KRG96b]:

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

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 edge is labeled with a dictionary 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 (bipartite) graph, whose left vertices are the original vertices of the graph, and whose right vertices represent the edges. 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 bipartite graph.

INPUT:

- `g` – Graph or DiGraph
- `partition` – (default:None) if given, the partition of the vertices is as well relabeled
- `standard_label` – (default:None) the standard label is not considered to be changed
- `return_relabeling` – (defaut:False) if True, a dictionary containing the relabeling is returned
- `return_edge_labels` – (defaut:False) if True, the different edge_labels are returned (useful if inplace is True)
- `inplace` – (default:False) if True, g is modified, otherwise the result is returned. Note that attributes of g are *not* copied for speed issues, only edges and vertices.

OUTPUT:

- if not inplace: the unlabeled graph without multiple edges
- the partition of the vertices
- if return_relabeling: a dictionary containing the relabeling
- if return_edge_labels: 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: g = graph_isom_equivalent_non_edge_labeled_graph(G, partition=[[0,123],[1,2]]); g
[Graph on 6 vertices, [[0, 3], [1, 2], [4], [5]]]

sage: g = graph_isom_equivalent_non_edge_labeled_graph(G); g
[Graph on 6 vertices, [[0, 1, 2, 3], [4], [5]]]
sage: g[0].edges()
[(0, 4, None), (1, 4, None), (1, 5, None), (2, 3, None), (2, 5, None)]

sage: g = graph_isom_equivalent_non_edge_labeled_graph(G,standard_label='string',return_edge_lab
[Graph on 6 vertices, [[0, 1, 2, 3], [5], [4]], [[[None, 1]], [[0, 1], [1, 1], [2, 1], [3, 1], [
sage: g[0].edges()
[(0, 4, None), (1, 2, None), (1, 4, None), (2, 5, None), (3, 5, None)]

sage: graph_isom_equivalent_non_edge_labeled_graph(G,inplace=True)
[[[0, 1, 2, 3], [4], [5]]]
sage: G.edges()
[(0, 4, None), (1, 4, None), (1, 5, None), (2, 3, None), (2, 5, None)]
```

Ensure that #14108 is fixed:

```
sage: G=DiGraph([[0,0],[0,0],[0,0],[1,1],[1,1],[1,1]])
sage: H=DiGraph([[0,0],[0,0],[0,0],[0,0],[1,1],[1,1]])
sage: G.is_isomorphic(H)
False
sage: H=DiGraph([[0,0],[0,0],[0,0],[0,0],[0,0],[1,1],[1,1]])
sage: HH=DiGraph([[0,0],[0,0],[0,0],[0,0],[1,1],[1,1],[1,1]])
sage: H.is_isomorphic(HH)
False
sage: H.is_isomorphic(HH, edge_labels=True)
False
```

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'>
sage: type(p)
<type 'dict'>
```

1.2 Undirected graphs

This module implements functions and operations involving undirected graphs.

Graph basic operations:

write_to_eps()	Writes a plot of the graph to filename in eps format.
to_undirected()	Since the graph is already undirected, simply returns a copy of itself.
to_directed()	Returns a directed version of the graph.
sparse6_string()	Returns the sparse6 representation of the graph as an ASCII string.
graph6_string()	Returns the graph6 representation of the graph as an ASCII string.
bipartite_sets()	Returns (X, Y) where X and Y are the nodes in each bipartite set of graph.
bipartite_color()	Returns a dictionary with vertices as the keys and the color class as the values.
is_directed()	Since graph is undirected, returns False.
join()	Returns the join of self and other.

Distances:

centrality_closeness()	Returns the closeness centrality (1/average distance to all vertices)
centrality_degree()	Returns the degree centrality
centrality_betweenness()	Returns the betweenness centrality

Graph properties:

is_prime()	Tests whether the current graph is prime.
is_split()	Returns True if the graph is a Split graph, False otherwise.
is_triangle_free()	Returns whether self is triangle-free.
is_bipartite()	Returns True if graph G is bipartite, False if not.
is_line_graph()	Tests wether the graph is a line graph.
is_odd_hole_free()	Tests whether self contains an induced odd hole.
is_even_hole_free()	Tests whether self contains an induced even hole.
is_cartesian_product()	Tests whether self is a cartesian product of graphs.
is_long_hole_free()	Tests whether self contains an induced cycle of length at least 5.
is_long_antihole_free()	Tests whether self contains an induced anticycle of length at least 5.
is_weakly_chordal()	Tests whether self is weakly chordal.
is_strongly_regular()	Tests whether self is strongly regular.
is_distance_regular()	Tests whether self is distance-regular.
is_tree()	Return True if the graph is a tree.
is_forest()	Return True if the graph is a forest, i.e. a disjoint union of trees.
is_overfull()	Tests whether the current graph is overfull.
odd_girth()	Returns the odd girth of self.
is_edge_transitive()	Returns true if self is edge-transitive.
is_arc_transitive()	Returns true if self is arc-transitive.
is_half_transitive()	Returns true if self is a half-transitive graph.
is_semi_symmetric()	Returns true if self is a semi-symmetric graph.

Connectivity, orientations, trees:

gomory_hu_tree()	Returns a Gomory-Hu tree of self.
minimum_outdegree_orientation()	Returns an orientation of self with the smallest possible maximum outdegree
bounded_outdegree_orientation()	Computes an orientation of self such that every vertex v has out-degree less than $b(v)$
strong_orientation()	Returns a strongly connected orientation of the current graph.
degree_constrained_subgraph()	Returns a degree-constrained subgraph.
bridges()	Returns the list of all bridges.
spanning_trees()	Returns the list of all spanning trees.

Clique-related methods:

clique_complex()	Returns the clique complex of self
cliques_containing_vertex()	Returns the cliques containing each vertex
cliques_vertex_clique_number()	Returns a dictionary of sizes of the largest maximal cliques containing each vertex
cliques_get_clique_bipartite()	Returns a bipartite graph constructed such that maximal cliques are the right vertices and the left vertices are retained from the given graph
cliques_get_max_clique_graph()	Returns a graph constructed with maximal cliques as vertices, and edges between maximal cliques sharing vertices.
cliques_number_of()	Returns a dictionary of the number of maximal cliques containing each vertex, keyed by vertex.
clique_number()	Returns the order of the largest clique of the graph.
clique_maximum()	Returns the vertex set of a maximal order complete subgraph.
cliques_maximum()	Returns the list of all maximum cliques
cliques_maximal()	Returns the list of all maximal cliques

Algorithmically hard stuff:

vertex_cover()	Returns a minimum vertex cover of self
independent_set()	Returns a maximum independent set.
topological_minor()	Returns a topological H-minor from self if one exists.
convexity_properties()	Returns a ConvexityProperties objet corresponding to self.
matching_polynomial()	Computes the matching polynomial of the graph G.
rank_decomposition()	Returns an rank-decomposition of self achieving optiml rank-width.
minor()	Returns the vertices of a minor isomorphic to H in the current graph.
independent_set_of_representatives()	Returns an independent set of representatives.
coloring()	Returns the first (optimal) proper vertex-coloring found.
has_homomorphism_to()	Checks whether there is a morphism between two graphs.
chromatic_number()	Returns the minimal number of colors needed to color the vertices of the graph.
chromatic_polynomial()	Returns the chromatic polynomial of the graph.
tutte_polynomial()	Returns the Tutte polynomial of the graph.
is_perfect()	Tests whether the graph is perfect.
treewidth()	Computes the tree-width and provides a decomposition.

Leftovers:

cores()	Returns the core number for each vertex in an ordered list.
matching()	Returns a maximum weighted matching of the graph
fractional_chromatic_index()	Computes the fractional chromatic index of self
kirchhoff_symanzik_polynomial()	Returns the Kirchhoff-Symanzik polynomial of the graph.
modular_decomposition()	Returns the modular decomposition of the current graph.
maximum_average_degree()	Returns the Maximum Average Degree (MAD) of the current graph.
two_factor_petersen()	Returns a decomposition of the graph into 2-factors.
ihara_zeta_function_inverse()	Returns the inverse of the zeta function of the graph.

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.

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, 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]
```

```
[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,-1,0,0,0,1,1,0,0,0,0,0),(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: 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
```

1.2.2 Generators

If you wish to iterate through all the isomorphism types of graphs, type, for example:

```
sage: for g in graphs(4):
...       print g.spectrum()
[0, 0, 0, 0]
[1, 0, 0, -1]
[1.4142135623..., 0, 0, -1.4142135623...]
[2, 0, -1, -1]
[1.7320508075..., 0, 0, -1.7320508075...]
[1, 1, -1, -1]
[1.6180339887..., 0.6180339887..., -0.6180339887..., -1.6180339887...]
[2.1700864866..., 0.3111078174..., -1, -1.4811943040...]
[2, 0, 0, -2]
```

```
[2.5615528128..., 0, -1, -1.5615528128...]
[3, -1, -1, -1]
```

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(immutabl
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*, *boundary=None*, *weighted=None*, *implementation='c_graph'*, *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 also the Wikipedia article on graphs.

One can very easily create a graph in Sage by typing:

```
sage: g = Graph()
```

By typing the name of the graph, one can get some basic information about it:

```
sage: g
Graph on 0 vertices
```

This graph is not very interesting as it is by default the empty graph. But Sage contains a large collection of pre-defined graph classes that can be listed this way:

- Within a Sage session, type `graphs.` (Do not press "Enter", and do not forget the final period ".")
- Hit "tab".

You will see a list of methods which will construct named graphs. For example:

```
sage: g = graphs.PetersenGraph()
sage: g.plot()
```

or:

```
sage: g = graphs.ChvatalGraph()
sage: g.plot()
```

In order to obtain more information about these graph constructors, access the documentation using the command `graphs.RandomGNP?`.

Once you have defined the graph you want, you can begin to work on it by using the almost 200 functions on graphs in the Sage library! If your graph is named `g`, you can list these functions as previously this way

- Within a Sage session, type `g.` (Do not press "Enter", and do not forget the final period ".")
- Hit "tab".

As usual, you can get some information about what these functions do by typing (e.g. if you want to know about the `diameter()` method) `g.diameter?`.

If you have defined a graph `g` having several connected components (i.e. `g.is_connected()` returns False), you can print each one of its connected components with only two lines:

```
sage: for component in g.connected_components():
...       g.subgraph(component).plot()
```

INPUT:

- `data` – can be any of the following (see the `format` argument):

1. An integer specifying the number of vertices
2. A dictionary of dictionaries
3. A dictionary of lists
4. A NumPy matrix or ndarray
5. A Sage adjacency matrix or incidence matrix
6. A pygraphviz graph
7. A SciPy sparse matrix
8. A NetworkX graph

- `pos` - 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` - (must be an explicitly named parameter, i.e., `name="complete")` gives the graph a name
- `loops` - boolean, whether to allow loops (ignored if data is an instance of the `Graph` class)
- `multiedges` - boolean, whether to allow multiple edges (ignored if data is an instance of the `Graph` class)

- weighted - whether graph thinks of itself as weighted or not. See self.weighted()
- format - if None, Graph tries to guess- can be several values, including:
  - 'int' - an integer specifying the number of vertices in an edge-free graph with vertices labelled from 0 to n-1
  - 'graph6' - Brendan McKay's graph6 format, in a string (if the string has multiple graphs, the first graph is taken)
  - 'sparse6' - Brendan McKay's sparse6 format, in a string (if the string has multiple graphs, the first graph is taken)
  - 'adjacency_matrix' - a square Sage matrix M, with M[i,j] equal to the number of edges {i,j}
  - 'weighted_adjacency_matrix' - a square Sage matrix M, with M[i,j] equal to the weight of the single edge {i,j}. Given this format, weighted is ignored (assumed True).
  - 'incidence_matrix' - a Sage matrix, with one column C for each edge, where if C represents {i, j}, C[i] is -1 and C[j] is 1
  - 'elliptic_curve_congruence' - data must be an iterable container of elliptic curves, and the graph produced has each curve as a vertex (it's Cremona label) and an edge E-F labelled p if and only if E is congruent to F mod p
  - NX - data must be a NetworkX Graph.

  Note: As Sage's default edge labels is None while NetworkX uses {}, the {} labels of a NetworkX graph are automatically set to None when it is converted to a Sage graph. This behaviour can be overruled by setting the keyword convert_empty_dict_labels_to_None to False (it is True by default).
- boundary - a list of boundary vertices, if empty, graph is considered as a 'graph without boundary'
- implementation - what to use as a backend for the graph. Currently, the options are either 'networkx' or 'c_graph'
- sparse (boolean) – 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
  - "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).

  Only available when implementation == 'c_graph'
- immutable (boolean) – whether to create a immutable graph. Note that immutable=True is actually a shortcut for data_structure='static_sparse'. Set to False by default, only available when implementation='c_graph'
- vertex_labels - only for implementation == 'c_graph'. 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 NumPy matrix or ndarray:

```
sage: import numpy
sage: A = numpy.array([[0,1,1],[1,0,1],[1,1,0]])
sage: Graph(A)
Graph on 3 vertices
```

6.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):
...
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 or
```

7.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
[-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: Graph(M)
Graph on 6 vertices

sage: Graph(Matrix([[1],[1],[1]]))
Traceback (most recent call last):
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: There r
sage: Graph(Matrix([[1],[1],[0]]))
Traceback (most recent call last):
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: Each c

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: Non-symmetric or non-square matrix assumed to be an incidence matrix: Each c


sage: MA = Matrix([[1,2,0], [0,2,0], [0,0,1]])      # trac 9714
sage: MI = Graph(MA, format='adjacency_matrix').incidence_matrix(); MI
[-1 -1  0  0  0  1]
[ 1  1  0  1  1  0]
[ 0  0  1  0  0  0]
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)]
```

8.a 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')])
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
```

Note that in all cases, we copy the NetworkX structure.

```
sage: import networkx
sage: g = networkx.Graph({0:[1,2,3], 2:[4]})
sage: G = Graph(g, implementation='networkx')
sage: H = Graph(g, implementation='networkx')
sage: G._backend._nxg is H._backend._nxg
False
```

All these graphs are mutable and can thus not be used as a dictionary key:

```
sage: {G:1}[H]
Traceback (most recent call last):
...
TypeError: This graph is mutable, and thus not hashable. Create an immutable copy by `g.copy(imm
```

When providing the optional arguments `data_structure="static_sparse"` or `immutable=True` (both mean the same), then an immutable graph results. Note that this does not use the NetworkX data structure:

```
sage: G_imm = Graph(g, immutable=True)
sage: H_imm = Graph(g, data_structure='static_sparse')
sage: G_imm == H_imm == G == H
True
sage: hasattr(G_imm._backend, "_nxg")
False
sage: {G_imm:1}[H_imm]
1
```

bipartite_color()

Returns 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()

Returns (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()
(set([0, 2]), set([1, 3]))
sage: graphs.CycleGraph(5).bipartite_sets()
Traceback (most recent call last):
...
RuntimeError: Graph is not bipartite.
```

bounded_outdegree_orientation(*bound*)

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.

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
```

TESTS:

As previously for random graphs, but more intensively:

```
sage: for i in xrange(30):      # long time (up to 6s on sage.math, 2012)
...       g = graphs.RandomGNP(40, .4)
...       b = lambda v : ceil(g.degree(v)/2)
...       D = g.bounded_outdegree_orientation(b)
...       if not (
```

```
...             all( D.out_degree(v) <= b(v) for v in g ) or
...             D.size() != g.size()):
...            print "Something wrong happened"
```

bridges()

Returns a list of the bridges (or cut edges).

A bridge is an edge so that deleting it disconnects the graph.

Note: This method assumes the graph is connected.

EXAMPLES:

```
sage: g = 2*graphs.PetersenGraph()
sage: g.add_edge(1,10)
sage: g.is_connected()
True
sage: g.bridges()
[(1, 10, None)]
```

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

Returns the betweenness centrality (fraction of number of shortest paths that go through each vertex) as a dictionary keyed by vertices. The betweenness is normalized by default to be in range (0,1). This wraps NetworkX's implementation of the algorithm described in [Brandes2003].

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 or None (default None) - if set to an integer, use `k` node samples to estimate betweenness. Higher values give better approximations.
- `weight` - None or string. If set to a string, use that attribute of the nodes as weight. `weight = True` is equivalent to `weight = "weight"`
- `endpoints` - Boolean. If set to True it includes the endpoints in the shortest paths count

REFERENCE:

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_betweenness()
{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: (graphs.ChvatalGraph()).centrality_betweenness(
...      normalized=False)
{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()
{0: 0.16666666666666666, 1: 0.16666666666666666, 2: 0.0, 3: 0.0}
```

centrality_closeness(*v=None*)

Returns the closeness centrality (1/average distance to all vertices) as a dictionary of values keyed by vertex. The degree centrality is normalized 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. '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,' [Borgatti95].

INPUT:

- v - a vertex label (to find degree centrality of only one vertex)

REFERENCE:

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_closeness()
{0: 0.61111111111111..., 1: 0.61111111111111..., 2: 0.61111111111111..., 3: 0.61111111111111
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_closeness()
{0: 1.0, 1: 1.0, 2: 0.75, 3: 0.75}
sage: D.centrality_closeness(v=1)
1.0
```

centrality_degree(*v=None*)

Returns the degree centrality (fraction of vertices connected to) as a dictionary of values keyed by vertex. The degree centrality is normalized 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. Degree centrality measures the number of links incident upon a vertex.

INPUT:

- v - a vertex label (to find degree centrality of only one vertex)

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_degree()
{0: 0.36363636363636365, 1: 0.36363636363636365, 2: 0.36363636363636365, 3: 0.3636363636363
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_degree()
{0: 1.0, 1: 1.0, 2: 0.6666666666666666, 3: 0.6666666666666666}
sage: D.centrality_degree(v=1)
1.0
```

chromatic_number(*algorithm='DLX', verbose=0*)

Returns the minimal number of colors needed to color the vertices of the graph G.

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).
- `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 xrange(2,
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 xrange(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 xrange(4,6))
True
```

A snark has chromatic index 4 hence its line graph has chromatic number 4:

```
sage: graphs.FlowerSnark().line_graph().chromatic_number()
4
```

TESTS:

```
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: G.chromatic_number(algorithm="foo")
Traceback (most recent call last):
```

```
...
ValueError: The 'algorithm' keyword must be set to either 'DLX', 'MILP' or 'CP'.
```

chromatic_polynomial(*G*, *return_tree_basis=False*)

Computes 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 - 7
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
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 xrange(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 xrange(11) if P(i) > 0)) == G.chromatic_number()
True
```

clique_complex()

Returns 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)}

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'*)

Returns the vertex set of a maximal order complete subgraph.

INPUT:

- `algorithm` – the algorithm to be used :
  - If `algorithm = "Cliquer"` (default) - This wraps the C program Cliquer [NisOst2003].
  - 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.

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 [NisOst2003].

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
```

TESTS:

Wrong algorithm:

```
sage: C.clique_maximum(algorithm = "BFS")
Traceback (most recent call last):
```

```
...
NotImplementedError: Only 'MILP', 'Cliquer' and 'mcqd' are supported.
```

clique_number(*algorithm='Cliquer'*, *cliques=None*)
Returns the order of the largest clique of the graph (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"` - This wraps the C program Cliquer [NisOst2003].
  - If `algorithm = "networkx"` - This function is based on NetworkX's implementation of the Bron and Kerbosch Algorithm [BroKer1973].
  - 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"`.

ALGORITHM:

This function is based on Cliquer [NisOst2003] and [BroKer1973].

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 xrange(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 xrange(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 xrange(1
True
```

TESTS:

```
sage: g = graphs.PetersenGraph()
sage: g.clique_number(algorithm="MILP")
2
sage: for i in range(10):                                    # optional - mcqd
```

```
...       g = graphs.RandomGNP(15,.5)                                    # optional - mcqd
...       if g.clique_number() != g.clique_number(algorithm="mcqd"): # optional - mcqd
...           print "This is dead wrong !"                               # optional - mcqd
```

cliques(**args*, ***kwds*)

Deprecated: Use `cliques_maximal()` instead. See trac ticket #5739 for details.

cliques_containing_vertex(*vertices=None*, *cliques=None*)

Returns 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,
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,
sage: F = graphs.Grid2dGraph(2,3)
sage: X = F.cliques_containing_vertex()
sage: for v in sorted(X.iterkeys()):
...     print v, X[v]
(0, 0) [[(0, 1), (0, 0)], [(1, 0), (0, 0)]]
(0, 1) [[(0, 1), (0, 0)], [(0, 1), (0, 2)], [(0, 1), (1, 1)]]
(0, 2) [[(0, 1), (0, 2)], [(1, 2), (0, 2)]]
(1, 0) [[(1, 0), (0, 0)], [(1, 0), (1, 1)]]
(1, 1) [[(0, 1), (1, 1)], [(1, 2), (1, 1)], [(1, 0), (1, 1)]]
(1, 2) [[(1, 2), (0, 2)], [(1, 2), (1, 1)]]
sage: F.cliques_containing_vertex(vertices=[(0, 1), (1, 2)])
{(0, 1): [[(0, 1), (0, 0)], [(0, 1), (0, 2)], [(0, 1), (1, 1)]], (1, 2): [[(1, 2), (0, 2)],
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]]}
```

cliques_get_clique_bipartite(***kwds*)

Returns 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_si
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(*name=''*)

Returns a graph constructed with maximal cliques as vertices, and edges 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.

INPUT:

- `name` - The name of the new 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_si
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'*)

Returns the list of all maximal cliques, with each clique 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 [BroKer1973].

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 [BroKer1973].

REFERENCE:

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 [NisOst2003].

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]]
```

TEST:

```
sage: g = Graph()
sage: g.cliques_maximum()
```

```
[[]]
```

cliques_number_of(*vertices=None*, *cliques=None*)

Returns a dictionary of the number of 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:

- `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: X = F.cliques_number_of()
sage: for v in sorted(X.iterkeys()):
...     print v, X[v]
(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*)

Returns 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 [NisOst2003].
  - **`networkx` - This function is based on NetworkX's implementation** of the Bron and Kerbosch Algorithm [BroKer1973].

- 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: X = F.cliques_vertex_clique_number(algorithm="networkx")
sage: for v in sorted(X.iterkeys()):
...     print v, X[v]
(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*, *verbose=0*)

Returns 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 – (default: False) if True, return a dictionary which can easily be used for plotting.
- 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"); P
[[2, 1, 3], [0, 6, 5], [4]]
sage: P = G.coloring(algorithm="DLX"); P
[[1, 2, 3], [0, 5, 6], [4]]
sage: G.plot(partition=P)
sage: H = G.coloring(hex_colors=True, algorithm="MILP")
```

```
sage: for c in sorted(H.keys()):
...       print c, H[c]
#0000ff [4]
#00ff00 [0, 6, 5]
#ff0000 [2, 1, 3]
sage: H = G.coloring(hex_colors=True, algorithm="DLX")
sage: for c in sorted(H.keys()):
...       print c, H[c]
#0000ff [4]
#00ff00 [1, 2, 3]
#ff0000 [0, 5, 6]
sage: G.plot(vertex_colors=H)
```

TESTS:

```
sage: G.coloring(algorithm="foo")
Traceback (most recent call last):
...
ValueError: The 'algorithm' keyword must be set to either 'DLX' or 'MILP'.
```

convexity_properties()

Returns 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*)

Returns 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). [WPkcore]

[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 [BZ].

INPUT

•`k` (integer)

–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)

–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.

By default, `with_labels = False`.

See Also:

•Graph cores is also a notion related to graph homomorphisms. For this second meaning, see `Graph.has_homomorphism_to()`.

REFERENCE:

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())
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:
```

```
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=None*, *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-constained 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
```

fractional_chromatic_index(*verbose_constraints=0*, *verbose=0*)

Computes the fractional chromatic index of `self`

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 on 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:

- `verbose_constraints` – whether to display which constraints are being generated.
- `verbose` – level of verbosity required from the LP solver

Note: This implementation can be improved by computing matchings through a LP formulation, and not using the Python implementation of Edmonds' algorithm (which requires to copy the graph, etc). It may be more efficient to write the matching problem as a LP, as we would then just have to update the weights on the edges between each call to `solve` (and so avoiding the generation of all the constraints).

EXAMPLE:

The fractional chromatic index of a C_5 is $5/2$:

```
sage: g = graphs.CycleGraph(5)
sage: g.fractional_chromatic_index()
2.5
```

gomory_hu_tree (*method='FF'*)

Returns a Gomory-Hu tree of self.

Given a tree T with labeled edges representing capacities, it is very easy to determine the maximal 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 maximal flow between any two vertices is the same in G as in T. See the Wikipedia article on Gomory-Hu tree. Note that, in general, a graph admits more than one Gomory-Hu tree.

INPUT:

- `method` – There are currently two different implementations of this method :
  - If `method = "FF"` (default), a Python implementation of the Ford-Fulkerson algorithm is used.
  - If `method = "LP"`, the flow problems are solved using Linear Programming.

OUTPUT:

A graph with labeled edges

EXAMPLE:

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()

Returns the graph6 representation of the graph as an ASCII string. Only valid for simple (no loops, multiple edges) graphs on 0 to 262143 vertices.

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 on graph homomorphisms.

INPUT:

- `H` – the graph to which `self` should be sent.
- `core` (boolean) – 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",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 dictionnary associating a vertex of H to a vertex of G.

EXAMPLE:

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
```

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 Wikipedia article Ihara zeta function

ALGORITHM:

This is computed here using the determinant 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 [ScottStorm].

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:

independent_set(*algorithm='Cliquer'*, *value_only=False*, *reduction_rules=True*, *solver=None*, *verbosity=0*)

Returns 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 [NisOst2003].

    (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*)

Returns an independent set of representatives.

Given a graph G and 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 [AhaBerZiv07] 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 xrange(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
```

REFERENCE:

is_arc_transitive()

Returns true 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 the wikipedia article on arc-transitive graphs for more information.

See Also:

- `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_bipartite (*certificate=False*)

Returns `True` if graph G is bipartite, `False` if not.

Traverse the graph G with breadth-first-search and color nodes.

INPUT:

- `certificate` – whether to return a certificate (`False` by default). If set to `True`, the certificate returned in 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(100,100,0.7).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
```

is_cartesian_product (*g*, *certificate=False*, *relabeling=False*)

Tests whether the graph is a cartesian product.

INPUT:

- `certificate` (boolean) – 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) – 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.

  This is set to False by default.

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 !

EXAMPLE:

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: g.is_cartesian_product(g, relabeling = True)
(True, {0: (0, 0), 1: (0, 1), 2: (0, 2), 3: (0, 3),
        4: (0, 4), 5: (5, 0), 6: (5, 1), 7: (5, 2),
        8: (5, 3), 9: (5, 4), 10: (10, 0), 11: (10, 1),
        12: (10, 2), 13: (10, 3), 14: (10, 4), 15: (15, 0),
        16: (15, 1), 17: (15, 2), 18: (15, 3), 19: (15, 4),
        20: (20, 0), 21: (20, 1), 22: (20, 2), 23: (20, 3),
        24: (20, 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
```

TESTS:

Wagner's Graph (trac ticket #13599):

```
sage: g = graphs.WagnerGraph()
sage: g.is_cartesian_product()
False
```

is_directed()

Since graph is undirected, returns False.

EXAMPLES:

```
sage: Graph().is_directed()
False
```

is_distance_regular(*G*, *parameters=False*)

Tests 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 its associated wikipedia page.

INPUT:

- `parameters` (boolean) – if set to `True`, the function returns the pair `(b,c)` of lists of integers instead of `True` (see the definition above). Set to `False` by default.

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
```

TESTS:

```
sage: graphs.PathGraph(2).is_distance_regular(parameters = True)
([1, None], [None, 1])
sage: graphs.Tutte12Cage().is_distance_regular(parameters=True)
([3, 2, 2, 2, 2, 2, None], [None, 1, 1, 1, 1, 1, 3])
```

is_edge_transitive()

Returns true 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 the wikipedia article on edge-transitive graphs for more information.

See Also:

- 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 [ABCHRS08].

INPUT:

- `certificate` (boolean) – When `certificate = False`, this method only returns `True` or `False`. If `certificate = True`, the subgraph found is returned instead of `False`.

EXAMPLE:

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 !
```

TESTS:

Bug reported in trac ticket #9925, and fixed by trac ticket #9420:

```
sage: g = Graph(':SiBFGaCEF_@CE`DEGH`CEFGaCDGaCDEHaDEF`CEH`ABCDEF', loops=False, multiedges=
sage: g.is_even_hole_free()
False
sage: g.is_even_hole_free(certificate = True)
Subgraph of (): Graph on 4 vertices
```

Making sure there are no other counter-examples around

```
sage: t = lambda x : (Graph(x).is_forest() or
...          isinstance(Graph(x).is_even_hole_free(certificate = True),Graph))
sage: all( t(graphs.RandomBipartite(10,10,.5)) for i in range(100) )
True
```

REFERENCE:

is_forest(*certificate=False*, *output='vertex'*)

Tests if the graph is a forest, i.e. a disjoint union of trees.

INPUT:

- `certificate` (boolean) – 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 and `(False, cycle)` when it contains a cycle.
- `output` (`'vertex'` (default) or `'edge'`) – whether the certificate is given as a list of vertices or a list of edges.

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, [63, 62, 61, 60, 64])
```

is_half_transitive()

Returns true if self is a half-transitive graph.

A graph is is half-transitive if it is both vertex and edge transitive but not arc-transitive.

See the wikipedia article on half-transitive graphs for more information.

See Also:

- 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_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.

Todo

This method sequentially tests each of the forbidden subgraphs in order to know whether the graph is a line graph, which is a very slow method. It could eventually be replaced by root_graph() when this method will not require an exponential time to run on general graphs anymore (see its documentation for more information on this problem)... and if it can be improved to return negative certificates !

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)}
```

TESTS:

Disconnected graphs:

```
sage: g = 2*graphs.CycleGraph(3)
sage: gl = g.line_graph().relabel(inplace = False)
sage: new_g = gl.is_line_graph(certificate = True)[1]
sage: g.line_graph().is_isomorphic(gl)
True
```

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 $O(m^2)$ [NikolopoulosPalios07] (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
```

TESTS:

Further tests:

```
sage: g = Graph({0:[6,7],1:[7,8],2:[8,9],3:[9,10],4:[10,11],5:[11,6],6:[0,5,7],7:[0,1,6],8:[
sage: r,a = g.is_long_antihole_free(certificate=True)
sage: r
False
sage: a.complement().is_isomorphic( graphs.CycleGraph(9) )
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 $O(m^2)$ [NikolopoulosPalios07] (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
```

TESTS:

Another graph with vertices 2, ..., 8, 10:

```
sage: g = Graph({2:[3,8],3:[2,4],4:[3,8,10],5:[6,10],6:[5,7],7:[6,8],8:[2,4,7,10],10:[4,5,8]
sage: r,hole = g.is_long_hole_free(certificate=True)
sage: r
False
sage: hole
Subgraph of (): Graph on 5 vertices
sage: hole.is_isomorphic(graphs.CycleGraph(hole.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 [CRST06], it is not known whether testing for one of these two cases independently is polynomial too.

INPUT:

- `certificate` (boolean) – When `certificate = False`, this method only returns `True` or `False`. If `certificate = True`, the subgraph found is returned instead of `False`.

EXAMPLE:

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
```

REFERENCES:

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_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 [SPGT] 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) – whether to return a certificate (default : `False`)

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.

EXAMPLE:

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
```

TEST:

Check that trac ticket #13546 has been fixed:

```
sage: Graph(':FgGE@I@GxGs', loops=False, multiedges=False).is_perfect()
False
sage: g = Graph({0: [2, 3, 4, 5],
...              1: [3, 4, 5, 6],
...              2: [0, 4, 5, 6],
...              3: [0, 1, 5, 6],
...              4: [0, 1, 2, 6],
...              5: [0, 1, 2, 3],
...              6: [1, 2, 3, 4]})
sage: g.is_perfect()
False
```

REFERENCES:

TESTS:

```
sage: Graph(':Ab').is_perfect()
Traceback (most recent call last):
...
ValueError: This method is only defined for simple graphs, and yours is not one of them !
sage: g = Graph()
sage: g.allow_loops(True)
sage: g.add_edge(0,0)
sage: g.edges()
[(0, 0, None)]
sage: g.is_perfect()
Traceback (most recent call last):
...
ValueError: This method is only defined for simple graphs, and yours is not one of them !
```

is_prime()

Tests whether the current graph is prime. A graph is prime if all its modules are trivial (i.e. empty, all of the graph or singletons)– see $self.modular_decomposition?$.

EXAMPLE:

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_semi_symmetric()

Returns true if self is semi-symmetric.

A graph is semi-symmetric if it is regular, edge-transitve but not vertex-transitive.

See the wikipedia article on semi-symmetric graphs for more information.

See Also:

- 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 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" [GraphClasses] 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 caracterisation 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: sum([g.subgraph_search_count(H,induced=True) for H in [graphs.CycleGraph(4),graphs.Cyc
0
```

REFERENCES:

is_strongly_regular(*g*, *parameters=False*)

Tests whether `self` 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 Wikipedia article Strongly regular graph

INPUT:

- `parameters` (boolean) – 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)]).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)]).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

INPUT:

- `certificate` (boolean) – 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 and `(False, cycle)` when it contains a cycle. It returns `(False, None)` when the graph is not connected.
- `output` (`'vertex'` (default) or `'edge'`) – whether the certificate is given as a list of vertices or a list of edges.

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
```

The empty graph is not considered to be a tree:

```
sage: graphs.EmptyGraph().is_tree()
False
```

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')])
sage: G.is_tree(certificate=True)
(False, [1, 2])
sage: G.is_tree(certificate=True, output='edge')
(False, [(1, 2, 'a'), (2, 1, 'b')])
```

TESTS:

trac ticket #14434 is fixed:

```
sage: g = Graph({0:[1,4,5],3:[4,8,9],4:[9],5:[7,8],7:[9]})
sage: _,cycle = g.is_tree(certificate=True)
sage: g.size()
10
sage: g.add_cycle(cycle)
sage: g.size()
10
```

is_triangle_free(*algorithm='bitset'*)

Returns whether `self` is triangle-free

INPUT:

- `algorithm` – (default: `'bitset'`) 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 generaly faster than stantard matrix multiplication.

EXAMPLE:

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
```

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
```

TESTS:

Comparison of algorithms:

```
sage: for i in xrange(10): # long test
...       G = graphs.RandomBarabasiAlbert(50,2)
...       bm = G.is_triangle_free(algorithm='matrix')
...       bb = G.is_triangle_free(algorithm='bitset')
...       if bm != bb:
...          print "That's not good!"
```

Asking for an unknown algorithm:: sage: g.is_triangle_free(algorithm='tip top') Traceback (most recent call last): ... ValueError: Algorithm 'tip top' not yet implemented. Please contribute.

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 = len(s.vertices())
```

```
sage: s.is_isomorphic( graphs.CycleGraph(1) )
True
```

join (*other*, *verbose_relabel=True*)

Returns the join of self and other.

INPUT:

- verbose_relabel - (defaults to True) If True, each vertex v in the first graph will be named '0,v' and each vertex u in the second graph will be named'1,u' in the final graph. If False, the vertices of the first graph and the second graph will be relabeled with consecutive integers.

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, verbose_relabel=False); 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,

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
sage: J.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
sage: J = G.join(H, verbose_relabel=False); 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 [Marcolli2009].

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')])
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')])
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:

matching(*value_only=False*, *algorithm='Edmonds'*, *use_edge_labels=True*, *solver=None*, *verbose=0*)

Returns a maximum weighted matching of the graph represented by the list of its edges. For more information, see the Wikipedia article on matchings.

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. 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. Only useful when `algorithm == "LP"`.

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)
9.0
```

Same test with the Linear Program formulation:

```
sage: g = graphs.PappusGraph()
sage: g.matching(algorithm="LP", value_only=True)
9.0
```

TESTS:

If `algorithm` is set to anything different from `"Edmonds"` or `"LP"`, an exception is raised:

```
sage: g = graphs.PappusGraph()
sage: g.matching(algorithm="somethingdifferent")
Traceback (most recent call last):
...
ValueError: algorithm must be set to either "Edmonds" or "LP"
```

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 [Godsil93]:

$$\mu(x) = \sum_{k \geq 0} (-1)^k p(G, k) x^{n-2k}$$

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 [Godsil93]:

- 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 [Godsil93] 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
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))
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)
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
```

TESTS:

Non-integer labels should work, (trac ticket #15545):

```
sage: G = Graph(10);
sage: G.add_vertex((0,1))
sage: G.add_vertex('X')
sage: G.matching_polynomial()
x^12
```

maximum_average_degree (*value_only=True*, *solver=None*, *verbose=0*)

Returns 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) – `True` by default
  - 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
```

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, is 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 [AMOZ06].

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.

EXAMPLE:

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
```

REFERENCES:

minor(*H*, *solver=None*, *verbose=0*)

Returns 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 on graph minor.

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()

Returns the modular decomposition of the current graph.

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 Parallel 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 "Serie," see the Wikipedia article on 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" [HabPau10].

OUTPUT:

A pair of two values (recursively encoding the decomposition) :

- The type of the current module :
  - `"Parallel"`
  - `"Prime"`
  - `"Serie"`
- The list of submodules (as list of pairs `(type, list)`, recursively...) or the vertex's name if the module is a singleton.

EXAMPLES:

The Bull Graph is prime:

```
sage: graphs.BullGraph().modular_decomposition()
('Prime', [3, 4, 0, 1, 2])
```

The Petersen Graph too:

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

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)
sage: g.modular_decomposition()
('Serie', [0, ('Parallel', [5, ('Serie', [1, 4, 3, 2]), 6])])
```

ALGORITHM:

This function uses a C implementation of a 2-step algorithm implemented by Fabien de Montgolfier [FMDec] :

- Computation of a factorizing permutation [HabibViennot1999].
- Computation of the tree itself [CapHabMont02].

See Also:

- `is_prime()` – Tests whether a graph is prime.

REFERENCE:

odd_girth()

Returns the odd girth of self.

The odd girth of a graph is defined as the smallest cycle of odd length.

OUTPUT:

The odd girth of `self`.

EXAMPLES:

The McGee graph has girth 7 and therefore its odd girth is 7 as well.

```
sage: G = graphs.McGeeGraph()
sage: G.odd_girth()
7
```

Any complete graph on more than 2 vertices contains a triangle and has thus odd girth 3.

```
sage: G = graphs.CompleteGraph(10)
sage: G.odd_girth()
3
```

Every bipartite graph has no odd cycles and consequently odd girth of infinity.

```
sage: G = graphs.CompleteBipartiteGraph(100,100)
sage: G.odd_girth()
+Infinity
```

See Also:

- `girth()` – computes the girth of a graph.

REFERENCES:

The property relating the odd girth to the coefficients of the characteristic polynomial is an old result from algebraic graph theory see

TESTS:

```
sage: graphs.CycleGraph(5).odd_girth()
5
sage: graphs.CycleGraph(11).odd_girth()
11
```

rank_decomposition(*G*, *verbose=False*)

Computes 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) – 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).

EXAMPLE:

```
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)
```

spanning_trees()

Returns a list of all spanning trees.

If the graph is disconnected, returns the empty list.

Uses the Read-Tarjan backtracking algorithm [RT75].

EXAMPLES:

```
sage: G = Graph([(1,2),(1,2),(1,3),(1,3),(2,3),(1,4)])
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)])
sage: len(G.spanning_trees())
6
sage: G.spanning_trees_count()
6
```

See Also:

spanning_trees_count() – counts the number of spanning trees.

REFERENCES:

sparse6_string()

Returns 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.

EXAMPLES:

```
sage: G = graphs.BullGraph()
sage: G.sparse6_string()
':Da@en'

sage: G = Graph()
sage: G.sparse6_string()
```

```
':?'

sage: G = Graph(loops=True, multiedges=True,data_structure="sparse")
sage: Graph(':?',data_structure="sparse") == G
True
```

strong_orientation()

Returns a strongly connected orientation of the current graph. See also the Wikipedia article Strongly_connected_component.

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.

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).

EXAMPLE:

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
True
```

A multigraph also has a strong orientation

```
sage: g = Graph([(1,2),(1,2)])
sage: g.strong_orientation()
Multi-digraph on 2 vertices
```

to_directed(*implementation='c_graph'*, *data_structure=None*, *sparse=None*)

Returns a directed version of the graph. A single edge becomes two edges, one in each direction.

INPUT:

- `implementation` - string (default: 'networkx') the implementation goes here. Current options are only 'networkx' or 'c_graph'.

- `data_structure` – one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of `Graph` or `DiGraph`.
- `sparse` (boolean) – `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_partition()

Return the partition of connected components of `self`.

EXAMPLES:

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

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*)

Returns 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 (= obtained from H through arbitrary subdivision of its edges) as a subgraph.

For more information, see the $Wikipediaarticleongraphminor : wikipedia : `Minor_(graph_theory)$.

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*)

Computes the tree-width of G (and provides a decomposition)

INPUT:

- `k` (integer) – 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` – whether to return the tree-decomposition itself.

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)
Graph on 7 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
```

TESTS:

```
sage: g = graphs.PathGraph(3)
sage: g.treewidth()
1
sage: g = 2*graphs.PathGraph(3)
sage: g.treewidth()
1
sage: g.treewidth(certificate=True)
Graph on 6 vertices
sage: g.treewidth(2)
True
sage: g.treewidth(1)
True
sage: Graph(1).treewidth()
1
sage: Graph(0).treewidth()
0
sage: graphs.PetersenGraph().treewidth(k=2)
False
sage: graphs.PetersenGraph().treewidth(k=6)
True
sage: graphs.PetersenGraph().treewidth(certificate=True).is_tree()
True
sage: graphs.PetersenGraph().treewidth(k=3,certificate=True)
False
sage: graphs.PetersenGraph().treewidth(k=4,certificate=True)
Graph on 7 vertices
```

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)
```

TESTS:

Providing an external cache:

```
sage: cache = {}
sage: _ = graphs.RandomGNP(7,.5).tutte_polynomial(cache=cache)
sage: len(cache) > 0
True
```

two_factor_petersen()

Returns 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).

EXAMPLE:

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]})
```

vertex_cover(*algorithm='Cliquer'*, *value_only=False*, *reduction_rules=True*, *solver=None*, *verbosity=0*)

Returns 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 on 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 two 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.
  - 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 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
```

TESTS:

The two algorithms should return the same result:

```
sage: g = graphs.RandomGNP(10,.5)
sage: vc1 = g.vertex_cover(algorithm="MILP")
sage: vc2 = g.vertex_cover(algorithm="Cliquer")
sage: len(vc1) == len(vc2)
True
```

The cardinality of the vertex cover is unchanged when reduction rules are used. First for trees:

```
sage: for i in range(20):
...       g = graphs.RandomTree(20)
...       vc1_set = g.vertex_cover()
...       vc1 = len(vc1_set)
...       vc2 = g.vertex_cover(value_only = True, reduction_rules = False)
...       if vc1 != vc2:
...           print "Error :", vc1, vc2
...           print "With reduction rules :", vc1
...           print "Without reduction rules :", vc2
...           break
...       g.delete_vertices(vc1_set)
...       if g.size() != 0:
...           print "This thing is not a vertex cover !"
```

Then for random GNP graphs:

```
sage: for i in range(20):
...       g = graphs.RandomGNP(50,4/50)
...       vc1_set = g.vertex_cover()
...       vc1 = len(vc1_set)
...       vc2 = g.vertex_cover(value_only = True, reduction_rules = False)
...       if vc1 != vc2:
...           print "Error :", vc1, vc2
...           print "With reduction rules :", vc1
...           print "Without reduction rules :", vc2
...           break
...       g.delete_vertices(vc1_set)
...       if g.size() != 0:
...           print "This thing is not a vertex cover !"
```

Testing mcqd:

```
sage: graphs.PetersenGraph().vertex_cover(algorithm="mcqd",value_only=True) # optional - mcq
6
```

Given a wrong algorithm:

```
sage: graphs.PetersenGraph().vertex_cover(algorithm = "guess")
Traceback (most recent call last):
...
ValueError: The algorithm must be "Cliquer" "MILP" or "mcqd".
```

REFERENCE:

write_to_eps (*filename*, ***options*)

Writes 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. `\usepackagegraphics` must appear in the preamble, and `\includegraphics{filename.eps}` will include the file. To compile the document to `pdf` with `pdflatex` the file needs first to be converted to `pdf`, for example with `ps2pdf filename.eps filename.pdf`.

`sage.graphs.graph.`**`compare_edges`**(*x*, *y*)

This function has been deprecated.

Compare edge x to edge y, return -1 if x y, 1 if x y, else 0.

TEST:

```
sage: G = graphs.PetersenGraph()
sage: E = G.edges()
sage: from sage.graphs.graph import compare_edges
sage: compare_edges(E[0], E[2])
doctest:...: DeprecationWarning: compare_edges(x,y) is deprecated.  Use statement 'cmp(x[1],y[1]
See http://trac.sagemath.org/13192 for details.
-1
```

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()`	Computes a (dummy) ranked layout so that all edges point upward.
`layout_acyclic()`	Computes a ranked layout so that all edges point upward.
`reverse()`	Returns a copy of digraph with edges reversed in direction.
`reverse_edge()`	Reverses an edge.
`reverse_edges()`	Reverses 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()`	Returns the list of the out-neighbors of a given vertex.
`neighbor_out_iterator()`	Returns an iterator over the out-neighbors of a given vertex.
`neighbors_in()`	Returns the list of the in-neighbors of a given vertex.
`neighbor_in_iterator()`	Returns an iterator over the in-neighbors of vertex.
`outgoing_edges()`	Returns a list of edges departing from vertices.
`outgoing_edge_iterator()`	Return an iterator over all departing edges from vertices
`incoming_edges()`	Returns a list of edges arriving at vertices.
`incoming_edge_iterator()`	Return an iterator over all arriving edges from vertices
`sources()`	Returns the list of all sources (vertices without incoming edges) of this digraph.
`sinks()`	Returns the list of all sinks (vertices without outoing edges) of this digraph.
`to_undirected()`	Returns 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()`	Returns the dig6 representation of the digraph as an ASCII string.

Paths and cycles:

all_paths_iterator()	Returns an iterator over the paths of self. The paths are
all_simple_paths()	Returns a list of all the simple paths of self starting
all_cycles_iterator()	Returns an iterator over all the cycles of self starting
all_simple_cycles()	Returns a list of all simple cycles of self.

Representation theory:

path_semigroup()	Returns the (partial) semigroup formed by the paths of the digraph.

Connectivity:

is_strongly_connected()	Returns whether the current DiGraph is strongly connected.
strongly_connected_components_digraph()	Returns the digraph of the strongly connected components
strongly_connected_components_subgraphs()	Returns the strongly connected components as a list of subgraphs.
strongly_connected_component_containing_vertex()	Returns the strongly connected component containing a given vertex
strongly_connected_components()	Returns the list of strongly connected components.

Acyclicity:

is_directed_acyclic()	Returns whether the digraph is acyclic or not.
is_transitive()	Returns whether the digraph is transitive or not.
is_aperiodic()	Returns whether the digraph is aperiodic or not.
level_sets()	Returns the level set decomposition of the digraph.
topological_sort_generator()	Returns a list of all topological sorts of the digraph if it is acyclic
topological_sort()	Returns a topological sort of the digraph if it is acyclic

Hard stuff:

feedback_edge_set()	Computes the minimum feedback edge (arc) set of a digraph

1.3.1 Methods

class sage.graphs.digraph.**DiGraph**(*data=None*, *pos=None*, *loops=None*, *format=None*, *boundary=None*, *weighted=None*, *implementation='c_graph'*, *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. For more information, see the Wikipedia article on digraphs.

One can very easily create a directed graph in Sage by typing:

```
sage: g = DiGraph()
```

By typing the name of the digraph, one can get some basic information about it:

```
sage: g
Digraph on 0 vertices
```

This digraph is not very interesting as it is by default the empty graph. But Sage contains several pre-defined digraph classes that can be listed this way:

- Within a Sage sessions, type `digraphs.` (Do not press "Enter", and do not forget the final period ".")
- Hit "tab".

You will see a list of methods which will construct named digraphs. For example:

```
sage: g = digraphs.ButterflyGraph(3)
sage: g.plot()
```

You can also use the collection of pre-defined graphs, then create a digraph from them.

```
sage: g = DiGraph(graphs.PetersenGraph())
sage: g.plot()
```

Calling `Digraph` on a graph returns the original graph in which every edge is replaced by two different edges going toward opposite directions.

In order to obtain more information about these digraph constructors, access the documentation by typing `digraphs.RandomDirectedGNP?`.

Once you have defined the digraph you want, you can begin to work on it by using the almost 200 functions on graphs and digraphs in the Sage library! If your digraph is named `g`, you can list these functions as previously this way

- Within a Sage session, type `g.` (Do not press "Enter", and do not forget the final period ".")
- Hit "tab".

As usual, you can get some information about what these functions do by typing (e.g. if you want to know about the `diameter()` method) `g.diameter?`.

If you have defined a digraph `g` having several connected components (i.e. `g.is_connected()` returns False), you can print each one of its connected components with only two lines:

```
sage: for component in g.connected_components():
...       g.subgraph(component).plot()
```

The same methods works for strongly connected components

```
sage: for component in g.strongly_connected_components():
...       g.subgraph(component).plot()
```

INPUT:

- `data` - can be any of the following (see the `format` keyword):
  1. A dictionary of dictionaries
  2. A dictionary of lists
  3. A numpy matrix or ndarray
  4. A Sage adjacency matrix or incidence matrix
  5. A pygraphviz graph
  6. A SciPy sparse matrix
  7. A NetworkX digraph
- `pos` - 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` - (must be an explicitly named parameter, i.e., name="complete") gives the graph a name
- `loops` - boolean, whether to allow loops (ignored if data is an instance of the DiGraph class)
- `multiedges` - boolean, whether to allow multiple edges (ignored if data is an instance of the DiGraph class)
- `weighted` - whether digraph thinks of itself as weighted or not. See self.weighted()
- `format` - if None, DiGraph tries to guess- can be several values, including:
  - `'adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the number of edges {i,j}
  - `'incidence_matrix'` - a Sage matrix, with one column C for each edge, where if C represents {i, j}, C[i] is -1 and C[j] is 1
  - `'weighted_adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the weight of the single edge {i,j}. Given this format, weighted is ignored (assumed True).
  - `NX` - data must be a NetworkX DiGraph.

  Note: As Sage's default edge labels is `None` while NetworkX uses `{}`, the `{}` labels of a NetworkX digraph are automatically set to `None` when it is converted to a Sage graph. This behaviour can be overruled by setting the keyword `convert_empty_dict_labels_to_None` to `False` (it is `True` by default).
- `boundary` - a list of boundary vertices, if empty, digraph is considered as a 'graph without boundary'
- `implementation` - what to use as a backend for the graph. Currently, the options are either 'networkx' or 'c_graph'
- `sparse` (boolean) – `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
  - `"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).

  Only available when `implementation == 'c_graph'`
- `immutable` (boolean) – whether to create a immutable digraph. Note that `immutable=True` is actually a shortcut for `data_structure='static_sparse'`. Set to `False` by default, only available when `implementation='c_graph'`
- `vertex_labels` - only for implementation == 'c_graph'. 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:

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()
[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 numpy matrix or ndarray:

```
sage: import numpy
sage: A = numpy.array([[0,1,0],[1,0,0],[1,1,0]])
sage: DiGraph(A)
Digraph on 3 vertices
```

5. 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 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
```

•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
[-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
```

6.A c_graph implemented DiGraph can be constructed from a networkx implemented DiGraph:

```
sage: D = DiGraph({0:[1],1:[2],2:[0]}, implementation="networkx")
sage: E = DiGraph(D,implementation="c_graph")
sage: D == E
True
```

7.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
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 sh

sage: D = DiGraph("IRAaDCI'OWEOKcPWAo")
Traceback (most recent call last):
...
RuntimeError: The string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

8.A NetworkX XDiGraph:

```
sage: import networkx
sage: g = networkx.MultiDiGraph({0:[1,2,3], 2:[4]})
```

```
sage: DiGraph(g)
Digraph on 5 vertices
```

9.A NetworkX digraph:

```
sage: import networkx
sage: g = networkx.DiGraph({0:[1,2,3], 2:[4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

Note that in all cases, we copy the NetworkX structure.

```
sage: import networkx
sage: g = networkx.DiGraph({0:[1,2,3], 2:[4]})
sage: G = DiGraph(g, implementation='networkx')
sage: H = DiGraph(g, implementation='networkx')
sage: G._backend._nxg is H._backend._nxg
False
```

TESTS:

```
sage: DiGraph({0:[1,2,3], 2:[4]}).edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (2, 4, None)]
sage: DiGraph({0:(1,2,3), 2:(4,)}).edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (2, 4, None)]
sage: DiGraph({0:Set([1,2,3]), 2:Set([4])}).edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (2, 4, None)]
```

Get rid of mutable default argument for *boundary* (trac ticket #14794):

```
sage: D = DiGraph(boundary=None)
sage: D._boundary
[]
```

Demonstrate that digraphs using the static backend are equal to mutable graphs but can be used as dictionary keys:

```
sage: import networkx
sage: g = networkx.DiGraph({0:[1,2,3], 2:[4]})
sage: G = DiGraph(g, implementation='networkx')
sage: G_imm = DiGraph(G, data_structure="static_sparse")
sage: H_imm = DiGraph(G, data_structure="static_sparse")
sage: H_imm is G_imm
False
sage: H_imm == G_imm == G
True
sage: {G_imm:1}[H_imm]
1
sage: {G_imm: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)`
```

The error message states that one can also create immutable graphs by specifying the `immutable` optional argument (not only by `data_structure='static_sparse'` as above):

```
sage: J_imm = DiGraph(G, immutable=True)
sage: J_imm == G_imm
True
sage: type(J_imm._backend) == type(G_imm._backend)
True
```

all_cycles_iterator(*starting_vertices=None*, *simple=False*, *rooted=False*, *max_length=None*, *trivial=False*)

Returns 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) on vertices from which the cycles must start. If None, then all vertices of the graph can be starting points.
- `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 cycles. If set to None, then all lengths are allowed.
- `trivial` - boolean (default: False). If set to True, then the empty cycles are also enumerated.

OUTPUT:

iterator

Note: 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 it.next()
['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)
[]
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 it.next()
['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*)

Returns an iterator over the paths of self. The paths are enumerated in increasing length order.

INPUT:

- `starting_vertices` - iterable (default: None) on the vertices from which the paths must start. If None, then all vertices of the graph can be starting points.
- `ending_vertices` - iterable (default: None) on the 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. These 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.

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()
sage: for _ in range(7): print pi.next()
['a', 'a']
['a', 'b']
['b', 'c']
['c', 'd']
['d', 'c']
['a', 'a', 'a']
['a', 'a', 'b']
```

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): print pi.next()
['a', 'a']
['a', 'b']
['a', 'a', 'a']
['a', 'a', 'b']
['a', 'b', 'c']
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b'])
sage: for _ in range(5): print pi.next()
['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: list(pi)
[['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']]
```

Or simply bound the length of the enumerated paths:

```
sage: pi = g.all_paths_iterator(starting_vertices=['a'], ending_vertices=['b', 'c'], max_len
sage: list(pi)
[['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']]
```

By default, empty paths are not enumerated, but it may be parametrized:

```
sage: pi = g.all_paths_iterator(simple=True, trivial=True)
sage: list(pi)
[['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=False)
sage: list(pi)
[['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']]
```

all_simple_cycles(*starting_vertices=None*, *rooted=False*, *max_length=None*, *trivial=False*)

Returns a list of all simple cycles of self.

INPUT:

- `starting_vertices` - iterable (default: None) on vertices from which the cycles must start. If None, then all vertices of the graph can be starting points.
- `rooted` - boolean (default: False). If set to False, then equivalent cycles are merged into one single cycle (the one starting with minimum vertex). Two cycles are called equivalent if they differ only from their starting vertex (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 cycles. If set to None, then all lengths are allowed.
- `trivial` - boolean (default: False). If set to True, then the empty cycles 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]]
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*)

Returns 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) of vertices from which the paths must start. If None, then all vertices of the graph can be starting points.
- `ending_vertices` - iterable (default: None) on the 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.

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({'a' : ['a', 'b'], 'b' : ['c'], 'c' : ['d'], 'd' : ['c']}, loops=True)
sage: g.all_simple_paths()
[['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']]
```

One may compute all paths having specific starting and/or ending vertices:

```
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.all_simple_paths(max_length=2)
[['a', 'a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'c'],
 ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'c'],
 ['d', 'c', 'd']]
```

By default, empty paths are not enumerated, but this can be parametrized:

```
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']]
```

dig6_string()

Returns the dig6 representation of the digraph as an ASCII string. Valid for single (no multiple edges) digraphs on 0 to 262143 vertices.

EXAMPLES:

```
sage: D = DiGraph()
sage: D.dig6_string()
'?'
sage: D.add_edge(0,1)
sage: D.dig6_string()
'AO'
```

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

Computes 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 on feedback arc sets.

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) – whether to use constraint generation when solving the Mixed Integer Linear Program (default: `True`).
- `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 problem is solved using Linear Programming, in two different ways. The first one is to solve the

following formulation:

$$\text{Minimize : } \sum_{(u,v)\in G} b_{(u,v)}$$

$$\text{Such that :}$$

$$\forall (u,v) \in G, d_u - d_v + n \cdot b_{(u,v)} \geq 0$$

$$\forall u \in G, 0 \leq d_u \leq |G|$$

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 :

$$\text{Minimize : } \sum_{(u,v)\in G} b_{(u,v)}$$

$$\text{Such that :}$$

$$\forall C \text{ circuits } \subseteq G, \sum_{uv\in C} b_{(u,v)} \geq 1$$

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) = g.edge_iterator().next()
sage: (u,v) in feedback or (v,u) in feedback
True
```

TESTS:

Comparing with/without constraint generation. Also double-checks ticket trac ticket #12833:

```
sage: for i in range(20):
...       g = digraphs.RandomDirectedGNP(10,.3)
...       x = g.feedback_edge_set(value_only = True)
...       y = g.feedback_edge_set(value_only = True,
...                 constraint_generation = False)
...       if x != y:
...          print "Oh my, oh my !"
...          break
```

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: for i in D.in_degree_iterator():
...    print i
3
3
2
3
2
2
2
3
sage: for i in D.in_degree_iterator(labels=True):
...    print i
((0, 1), 3)
((1, 2), 3)
((0, 0), 2)
((0, 2), 3)
((1, 3), 2)
((1, 0), 2)
((0, 3), 2)
((1, 1), 3)
```

in_degree_sequence()

Return the indegree sequence.

EXAMPLES:

The indegree 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) – 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*)

Returns a list of edges arriving at vertices.

INPUT:

- vertices – a vertex or a list of vertices
- labels (boolean) – 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, cf. Wikipedia article Aperiodic_graph.

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
```

is_directed()

Since digraph is directed, returns True.

EXAMPLES:

```
sage: DiGraph().is_directed()
True
```

is_directed_acyclic(*certificate=False*)

Returns 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` – whether to return a certificate (`False` by default).

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],
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
```

TESTS:

What about loops?:

```
sage: g = digraphs.ButterflyGraph(3)
sage: g.allow_loops(True)
sage: g.is_directed_acyclic()
True
sage: g.add_edge(0,0)
sage: g.is_directed_acyclic()
False
```

is_strongly_connected()

Returns whether the current DiGraph is strongly connected.

EXAMPLE:

The circuit is obviously strongly connected

```
sage: g = digraphs.Circuit(5)
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
```

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 $uv \notin G$.

EXAMPLE:

```
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: digraphs.DeBruijn(5,2).is_transitive()
False
sage: digraphs.DeBruijn(5,2).is_transitive(certificate = True)
('00', '10')
```

```
sage: digraphs.RandomDirectedGNP(20,.2).transitive_closure().is_transitive()
True
```

layout_acyclic(***options*)

Computes 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 random horizontal placement of the vertices otherwise (see `layout_acyclic_dummy()`).

Non acyclic graphs are partially supported by `graphviz`, which then chooses some edges to point down.

EXAMPLES:

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

The actual layout computed will depend on whether dot2tex and graphviz are installed, so we don't test it.

sage: H.layout_acyclic() {0: [..., ...], 1: [..., ...], 2: [..., ...], 3: [..., ...], 5: [..., ...], 6: [..., ...]}

layout_acyclic_dummy(*heights=None*, ***options*)

Computes a (dummy) ranked layout of an acyclic graph 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()`).

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.

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):
...
AssertionError: `self` should be an acyclic graph
```

level_sets()

Returns 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*)

Returns 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*)

Returns an iterator over the out-neighbors of a given vertex.

An 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*)

Returns the list of the in-neighbors of a given 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: D.neighbors_in(0)
[1, 4]
```

neighbors_out(*vertex*)

Returns the list of the out-neighbors of a given vertex.

An 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_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: for i in D.out_degree_iterator():
...     print i
3
3
2
3
2
2
2
3
sage: for i in D.out_degree_iterator(labels=True):
...     print i
((0, 1), 3)
((1, 2), 3)
((0, 0), 2)
((0, 2), 3)
((1, 3), 2)
((1, 0), 2)
((0, 3), 2)
((1, 1), 3)
```

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) – 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
(0, 1, None)
(0, 2, None)
(0, 3, None)
```

outgoing_edges(*vertices*, *labels=True*)

Returns a list of edges departing from vertices.

INPUT:

- `vertices` – a vertex or a list of vertices
- `labels` (boolean) – 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]
```

predecessor_iterator(**args*, ***kwds*)

Deprecated: Use `neighbor_in_iterator()` instead. See trac ticket #7634 for details.

predecessors(**args*, ***kwds*)

Deprecated: Use `neighbors_in()` instead. See trac ticket #7634 for details.

reverse()

Returns 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*)

Reverses the edge from u to v.

INPUT:

- **`inplace` – (default: True) if `False`, a new digraph is created** and returned as output, otherwise `self` is modified.
- `multiedges` – (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, multiedge
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 chose 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()
[(2, 1, 'label'), ((1, 2), 'label', None)]
sage: D.reverse_edge((1,2), 'label')
sage: D.edges()
[(2, 1, 'label'), ('label', (1, 2), None)]
```

TESTS:

```
sage: D = DiGraph([(0,1,None)])
sage: D.reverse_edge(0,1,'mylabel')
Traceback (most recent call last):
...
ValueError: Input edge must exist in the digraph.
```

reverse_edges(*edges*, *inplace=True*, *multiedges=None*)

Reverses a list of edges.

INPUT:

- `edges` – a list of edges in the DiGraph.
- **`inplace` – (default: True) if `False`, a new digraph is created** and returned as output, otherwise `self` is modified.
- **`multiedges` – (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')]
```

TESTS:

```
sage: D = digraphs.Circuit(6)
sage: D.reverse_edges(D.edges(),inplace=False).edges()
[(0, 5, None), (1, 0, None), (2, 1, None),
 (3, 2, None), (4, 3, None), (5, 4, None)]

sage: D = digraphs.Kautz(2,3)
sage: Dr = D.reverse_edges(D.edges(),inplace=False,multiedges=True)
sage: Dr.edges() == D.reverse().edges()
True
```

sinks()

Returns a list of sinks of the digraph.

OUTPUT:

- list, 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()
Returns a list of sources of the digraph.

OUTPUT:

•list, 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]
```

strongly_connected_component_containing_vertex(*v*)
Returns the strongly connected component containing a given vertex

INPUT:

•v – a vertex

EXAMPLE:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: d.strongly_connected_component_containing_vertex(0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

strongly_connected_components()
Returns the list of strongly connected components.

EXAMPLES:

```
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()
[[0], [1], [2], [3], [4], [5], [6]]
sage: D.add_edge([2,0])
sage: D.strongly_connected_components()
[[0, 1, 2], [3], [4], [5], [6]]
```

TESTS:

Checking against NetworkX, and another of Sage's implementations:

```
sage: from sage.graphs.base.static_sparse_graph import strongly_connected_components
sage: import networkx
sage: for i in range(100):                                          # long
...       g = digraphs.RandomDirectedGNP(100,.05)                   # long
...       h = g.networkx_graph()                                    # long
...       scc1 = g.strongly_connected_components()                  # long
...       scc2 = networkx.strongly_connected_components(h)          # long
```

```
...         scc3 = strongly_connected_components(g)             # long
...         s1 = Set(map(Set,scc1))                             # long
...         s2 = Set(map(Set,scc2))                             # long
...         s3 = Set(map(Set,scc3))                             # long
...         if s1 != s2:                                        # long
...             print "Ooch !"                                  # long
...         if s1 != s3:                                        # long
...             print "Oooooch !"                               # long
```

strongly_connected_components_digraph(*keep_labels=False*)

Returns the digraph of the strongly connected components

INPUT:

- `keep_labels` – boolean (default: False)

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 from one to the other in G.

EXAMPLE:

Such a digraph is always acyclic

```
sage: g = digraphs.RandomDirectedGNP(15,.1)
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 = g.strongly_connected_components_digraph()
sage: g.is_directed_acyclic()
True
sage: all([ Set(scc) in scc_digraph.vertices() for scc in g.strongly_connected_components()]
True
```

The following digraph has three strongly connected components, and the digraph of those is a chain:

```
sage: g = DiGraph({0:{1:"01", 2: "02", 3: 03}, 1: {2: "12"}, 2:{1: "21", 3: "23"}})
sage: scc_digraph = g.strongly_connected_components_digraph()
sage: scc_digraph.vertices()
[{0}, {3}, {1, 2}]
sage: scc_digraph.edges()
[({0}, {3}, None), ({0}, {1, 2}, None), ({1, 2}, {3}, None)]
```

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",
sage: scc_digraph = g.strongly_connected_components_digraph(keep_labels = True)
sage: scc_digraph.vertices()
[{0}, {3}, {1, 2}]
sage: scc_digraph.edges()
[({0}, {3}, '0-3'), ({0}, {1, 2}, '0-12'),
 ({1, 2}, {3}, '1-3'), ({1, 2}, {3}, '2-3'),
 ({1, 2}, {1, 2}, '1-2'), ({1, 2}, {1, 2}, '2-1')]
```

strongly_connected_components_subgraphs()
Returns the strongly connected components as a list of subgraphs.

EXAMPLE:

In the symmetric digraph of a graph, the strongly connected components are the connected components:

```
sage: g = graphs.PetersenGraph()
sage: d = DiGraph(g)
sage: d.strongly_connected_components_subgraphs()
[Subgraph of (Petersen graph): Digraph on 10 vertices]
```

successor_iterator(**args*, ***kwds*)
Deprecated: Use `neighbor_out_iterator()` instead. See trac ticket #7634 for details.

successors(**args*, ***kwds*)
Deprecated: Use `neighbors_out()` instead. See trac ticket #7634 for details.

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(*implementation='c_graph'*, *data_structure=None*, *sparse=None*)
Returns an undirected version of the graph. Every directed edge becomes an edge.

INPUT:

- `implementation` - string (default: 'networkx') the implementation goes here. Current options are only 'networkx' or 'c_graph'.
- `data_structure` – one of `"sparse"`, `"static_sparse"`, or `"dense"`. See the documentation of `Graph` or `DiGraph`.
- `sparse` (boolean) – `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'*)
Returns a topological sort of the digraph if it is acyclic, and raises a TypeError if the digraph contains a directed cycle. 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` – Use the default Cython implementation (`implementation = default`), the default NetworkX library (`implementation = "NetworkX"`) or the recursive NetworkX implementation (`implementation = "recursive"`)

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 graph1).

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: D.topological_sort(implementation = "NetworkX")
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]
```

Using the NetworkX recursive implementation

```
sage: D.topological_sort(implementation = "recursive")
[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.
```

Note: There is a recursive version of this in NetworkX, it used to have problems in earlier versions but they have since been fixed:

```
sage: import networkx
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: N = D.networkx_graph()
sage: networkx.topological_sort(N)
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]
sage: networkx.topological_sort_recursive(N)
[4, 5, 6, 9, 0, 3, 2, 7, 1, 8, 10]
```

TESTS:

A wrong value for the `implementation` keyword:

```
sage: D.topological_sort(implementation = "cloud-reading")
Traceback (most recent call last):
...
ValueError: implementation must be set to one of "default"
or "NetworkX"
```

topological_sort_generator()

Returns a list of all topological sorts of the digraph if it is acyclic, and raises a TypeError if the digraph contains a directed cycle.

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 Graph.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: D.topological_sort_generator()
[[0, 1, 2, 3, 4], [0, 1, 2, 4, 3], [0, 2, 1, 3, 4], [0, 2, 1, 4, 3], [0, 2, 4, 1, 3]]

sage: for sort in D.topological_sort_generator():
...       for edge in D.edge_iterator():
...           u,v,l = edge
...           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

TESTS:

```
sage: B = graphs.CompleteBipartiteGraph(7, 9)
sage: loads(dumps(B)) == B
True

sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B == B.copy()
True
sage: type(B.copy())
<class 'sage.graphs.bipartite_graph.BipartiteGraph'>
```

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.

4.A file in alist format.

5.From a NetworkX bipartite 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]]`.

The alist file format is described at http://www.inference.phy.cam.ac.uk/mackay/codes/alist.html

- `partition` – (default: `None`) a tuple defining vertices of the left and right partition of the graph. Partitions will be determined automatically if `partition``=``None`.
- `check` – (default: `True`) if `True`, an invalid input partition raises an exception. In the other case offending edges simply won't be included.

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):
...
TypeError: 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
set([0, 1, 2, 3])
sage: B.right
set([4, 5, 6])
sage: B = BipartiteGraph({0:[5,6], 1:[4,5], 2:[4,6], 3:[4,5,6]})
sage: B == G
True
sage: B.left
set([0, 1, 2, 3])
sage: B.right
set([4, 5, 6])
```

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 = [range(5), 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
set([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, [range(4), range(4,7)])
sage: B3
Bipartite graph on 7 vertices
sage: B3 == B2
True
```

::

```
sage: G = Graph({0:[], 1:[], 2:[]})
sage: part = (range(2), [2])
sage: B = BipartiteGraph(G, part)
sage: B2 = BipartiteGraph(B)
sage: B == B2
True
```

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]
[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();
sage: B = BipartiteGraph(file_name)
sage: B == 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)
```

TESTS:

Make sure we can create a BipartiteGraph with keywords but no positional arguments (trac #10958).

```
sage: B = BipartiteGraph(multiedges=True)
sage: B.allows_multiple_edges()
True
```

Ensure that we can construct a BipartiteGraph with isolated vertices via the reduced adjacency matrix (trac #10356):

```
sage: a=BipartiteGraph(matrix(2,2,[1,0,1,0]))
sage: a
Bipartite graph on 4 vertices
sage: a.vertices()
[0, 1, 2, 3]
sage: g = BipartiteGraph(matrix(4,4,[1]*4+[0]*12))
sage: g.vertices()
[0, 1, 2, 3, 4, 5, 6, 7]
sage: sorted(g.left.union(g.right))
[0, 1, 2, 3, 4, 5, 6, 7]
```

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

Adds an edge from u and 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 Graph.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.

TEST:

```
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_edge(0,2)
Traceback (most recent call last):
...
RuntimeError: Edge vertices must lie in different partitions.
sage: bg.add_edge(0,3); list(bg.right)
[1, 3]
sage: bg.add_edge(5,6); 5 in bg.left; 6 in bg.right
```

```
True
True
```

add_vertex(*name=None*, *left=False*, *right=False*)

Creates 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` – (default: `False`) if `True`, puts the new vertex in the left partition.
- `right` – (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` ``!=`` `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
set([0])
sage: G.right
set([1])
```

TESTS:

Exactly one of `left` and `right` must be true:

```
sage: G = BipartiteGraph()
sage: G.add_vertex()
Traceback (most recent call last):
...
RuntimeError: Partition must be specified (e.g. left=True).
sage: G.add_vertex(left=True, right=True)
Traceback (most recent call last):
...
RuntimeError: Only one partition may be specified.
```

Adding the same vertex must specify the same partition:

```
sage: bg = BipartiteGraph()
sage: bg.add_vertex(0, right=True)
sage: bg.add_vertex(0, right=True)
sage: bg.vertices()
[0]
sage: bg.add_vertex(0, left=True)
Traceback (most recent call last):
```

```
...
RuntimeError: Cannot add duplicate vertex to other partition.
```

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.

INPUTS:

- `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
set([0, 1, 2, 3, 5, 7])
sage: bg.right
set([4, 6, 8, 9, 10, 11])
```

TEST:

```
sage: bg = BipartiteGraph()
sage: bg.add_vertices([0,1,2], left=True)
sage: bg.add_vertices([0,1,2], left=[True,True,True])
sage: bg.add_vertices([0,1,2], right=[False,False,False])
sage: bg.add_vertices([0,1,2], right=[False,False,False])
sage: bg.add_vertices([0,1,2])
Traceback (most recent call last):
...
RuntimeError: Partition must be specified (e.g. left=True).
sage: bg.add_vertices([0,1,2], left=True, right=True)
Traceback (most recent call last):
...
RuntimeError: Only one partition may be specified.
sage: bg.add_vertices([0,1,2], right=True)
Traceback (most recent call last):
...
RuntimeError: Cannot add duplicate vertex to other partition.
sage: (bg.left, bg.right)
(set([0, 1, 2]), set([]))
```

bipartition()

Returns the underlying bipartition of the bipartite graph.

EXAMPLE:

```
sage: B = BipartiteGraph(graphs.CycleGraph(4))
sage: B.bipartition()
(set([0, 2]), set([1, 3]))
```

delete_vertex(*vertex*, *in_order=False*)
Deletes vertex, removing all incident edges. Deleting a non-existent vertex will raise an exception.

INPUT:

- `vertex` – a vertex to delete.
- `in_order` – (default `False`) if `True`, this 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
set([2])
sage: B.edges()
[(1, 2, None), (2, 3, None)]
sage: B.delete_vertex(3)
sage: B.right
set([1])
sage: B.edges()
[(1, 2, None)]
sage: B.delete_vertex(0)
Traceback (most recent call last):
...
RuntimeError: 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
set([2])
sage: B.right
set([1])
sage: B.edges()
[(1, 2, None)]
sage: B.delete_vertices([0])
Traceback (most recent call last):
...
RuntimeError: Vertex (0) not in the graph.
```

load_afile(*fname*)

Loads 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.

EXAMPLE:

```
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),
 (4, 8, None),
 (5, 9, None),
 (6, 10, None)]
 sage: B2 = BipartiteGraph(file_name)
 sage: B2 == B
 True
```

plot(**args*, ***kwds*)

Overrides Graph's plot function, to illustrate the bipartite nature.

EXAMPLE:

```
sage: B = BipartiteGraph(graphs.CycleGraph(20))
sage: B.plot()
```

project_left()

Projects `self` onto left vertices. Edges are 2-paths in the original.

EXAMPLE:

```
sage: B = BipartiteGraph(graphs.CycleGraph(20))
sage: G = B.project_left()
sage: G.order(), G.size()
(10, 10)
```

project_right()

Projects `self` onto right vertices. Edges are 2-paths in the original.

EXAMPLE:

```
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]]`.

This method supports the named argument 'sparse' which defaults to `True`. When enabled, the returned matrix will be sparse.

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
```

TESTS:

```
sage: B = BipartiteGraph()
sage: B.reduced_adjacency_matrix()
[]
sage: M = Matrix([[0,0], [0,0]])
sage: BipartiteGraph(M).reduced_adjacency_matrix() == M
True
sage: M = Matrix([[10,2/3], [0,0]])
sage: B = BipartiteGraph(M, weighted=True, sparse=True)
sage: M == B.reduced_adjacency_matrix()
True
```

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.

EXAMPLE:

```
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 == b2
True
```

TESTS:

```
sage: file_name = os.path.join(SAGE_TMP, 'deleteme.alist.txt')
sage: for order in range(3, 13, 3):
....:     num_chks = int(order / 3)
....:     num_vars = order - num_chks
....:     partition = (range(num_vars), range(num_vars, num_vars+num_chks))
....:     for idx in range(100):
....:         g = graphs.RandomGNP(order, 0.5)
....:         try:
....:             b = BipartiteGraph(g, partition, check=False)
....:             b.save_afile(file_name)
....:             b2 = BipartiteGraph(file_name)
....:             if b != b2:
....:                 print "Load/save failed for code with edges:"
....:                 print b.edges()
....:                 break
....:         except Exception:
....:             print "Exception encountered for graph of order "+ str(order)
....:             print "with edges: "
```

```
....:             g.edges()
....:             raise
```

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
```

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

BullGraph	CompleteMultipartiteGraph	LadderGraph
ButterflyGraph	DiamondGraph	LollipopGraph
CircularLadderGraph	EmptyGraph	PathGraph
ClawGraph	Grid2dGraph	StarGraph
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	FosterGraph	MoebiusKantorGraph
Balaban11Cage	FranklinGraph	MoserSpindle
BidiakisCube	FruchtGraph	NauruGraph
BiggsSmithGraph	GoldnerHararyGraph	PappusGraph
BlanusaFirstSnarkGraph	GrayGraph	PoussinGraph
BlanusaSecondSnarkGraph	GrotzschGraph	PetersenGraph
BrinkmannGraph	HallJankoGraph	RobertsonGraph
BrouwerHaemersGraph	HarriesGraph	SchlaefliGraph
BuckyBall	HarriesWongGraph	ShrikhandeGraph
CameronGraph	HeawoodGraph	SimsGewirtzGraph
Cell600	HerschelGraph	SousselierGraph
Cell120	HigmanSimsGraph	SylvesterGraph
ChvatalGraph	HoffmanGraph	SzekeresSnarkGraph
ClebschGraph	HoffmanSingletonGraph	ThomsenGraph
CoxeterGraph	HoltGraph	TietzeGraph
DesarguesGraph	HortonGraph	Tutte12Cage
DoubleStarSnark	KittellGraph	TutteCoxeterGraph
DurerGraph	KrackhardtKiteGraph	TutteGraph
DyckGraph	LjubljanaGraph	WagnerGraph
EllinghamHorton54Graph	M22Graph	WatkinsSnarkGraph
EllinghamHorton78Graph	MarkstroemGraph	WellsGraph
ErreraGraph	McGeeGraph	WienerArayaGraph
FlowerSnark	McLaughlinGraph	
FolkmanGraph	MeredithGraph	

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 ...)

AffineOrthogonalPolarGraph	fusenes	NKStarGraph
BalancedTree	FuzzyBallGraph	NStarGraph
BarbellGraph	GeneralizedPetersenGraph	OddGraph
BubbleSortGraph	HanoiTowerGraph	OrthogonalPolarGraph
CirculantGraph	HararyGraph	PaleyGraph
cospectral_graphs	HyperStarGraph	petersen_family
CubeGraph	JohnsonGraph	RingedTree
DorogovtsevGoltsevMendesGraph	KneserGraph	SymplecticGraph
FibonacciTree	LCFGraph	trees
FoldedCubeGraph	line_graph_forbidden_subgraphs	WheelGraph
FriendshipGraph	MycielskiGraph	
fullerenes	MycielskiStep	

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	RandomHolmeKim	RandomShell
RandomBipartite	RandomIntervalGraph	RandomToleranceGraph
RandomBoundedToleranceGraph	RandomLobster	RandomTree
RandomGNM	RandomNewmanWattsStrogatz	RandomTreePowerlaw
RandomGNP	RandomRegular	

Graphs with a given degree sequence

DegreeSequence	DegreeSequenceConfigurationModel	DegreeSequenceTree
DegreeSequenceBipartite	DegreeSequenceExpected	

Miscellaneous

WorldMap		

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
- 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

2.1.1 Functions and methods

class sage.graphs.graph_generators.**GraphGenerators**

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 optional nauty package for generating graphs at the nauty_geng() method.

INPUT:

- vertices – natural number.
- 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.
- implementation – (default: 'c_graph') which underlying implementation to use (see Graph?).
- sparse – (default: True) ignored if implementation is not 'c_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_graph = 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
```

Note that we can also get graphs with underlying Cython implementation:

```
sage: for G in graphs(3, augment='vertices', implementation='c_graph'):
...     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 http://oeis.org/classic/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: len(filter(property, graphs(4)))
4
sage: property = lambda G: G.is_bipartite()
sage: len(list(graphs(4, property)))
7
sage: len(filter(property, graphs(4)))
7
```

Generate graphs on the fly: (see http://oeis.org/classic/A000088)

```
sage: for i in range(0, 7):
...     print len(list(graphs(i)))
1
1
2
4
11
34
156
```

Generate all simple graphs, allowing loops: (see http://oeis.org/classic/A000666)

```
sage: L = list(graphs(5,augment='vertices',loops=True))               # long time
sage: for i in [0..5]: 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 http://oeis.org/classic/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_
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 te
10 19
```

Make sure that the graphs are really independent and the generator survives repeated vertex removal (trac 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 != None`, and odd otherwise.
- `q` (integer) – a power of a prime number, as F_q must exist.
- `sign` – must be qual 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 `Balaban10Cage`(*embedding=1*)

Returns the Balaban 10-cage.

The Balaban 10-cage is a 3-regular graph with 70 vertices and 105 edges. See its Wikipedia page.

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
```

TESTS:

```
sage: graphs.Balaban10Cage(embedding='foo')
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

static Balaban11Cage(*embedding=1*)

Returns the Balaban 11-cage.

For more information, see this Wikipedia article on the 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 [FAGDC]. 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
```

TESTS:

```
sage: graphs.Balaban11Cage(embedding='xyzzy')
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1, 2, or 3.
```

REFERENCES:

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()
True
sage: v = (r^(h + 1) - 1) / (r - 1)
sage: T.order() == v
True
sage: T.size() == v - 1
True
```

TESTS:

Normally we would only consider balanced trees whose root node has degree $r \geq 2$, but the construction degenerates gracefully:

```
sage: graphs.BalancedTree(1, 10)
Balanced tree: Graph on 2 vertices

sage: graphs.BalancedTree(-1, 10)
Balanced tree: Graph on 1 vertex
```

Similarly, we usually want the tree must have height $h \geq 1$ but the algorithm also degenerates gracefully here:

```
sage: graphs.BalancedTree(3, 0)
Balanced tree: Graph on 1 vertex

sage: graphs.BalancedTree(5, -2)
Balanced tree: Graph on 0 vertices

sage: graphs.BalancedTree(-2,-2)
Balanced tree: Graph on 0 vertices
```

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.

This constructor depends on NetworkX numeric labels. In this case, the `n1`-th node connects to the path graph from one complete graph and the `n1 + n2 + 1`-th node connects to the path graph from the other complete graph.

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`.

ALGORITHM:

Uses NetworkX.

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
```

Create several barbell graphs in a Sage graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(6):
...       k = graphs.BarbellGraph(i + 2, 4)
...       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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

TESTS:

The input `n1` must be ≥ 2:

```
sage: graphs.BarbellGraph(1, randint(0, 10^6))
Traceback (most recent call last):
...
ValueError: Invalid graph description, n1 should be >= 2
sage: graphs.BarbellGraph(randint(-10^6, 1), randint(0, 10^6))
Traceback (most recent call last):
...
ValueError: Invalid graph description, n1 should be >= 2
```

The input `n2` must be ≥ 0:

```
sage: graphs.BarbellGraph(randint(2, 10^6), -1)
Traceback (most recent call last):
...
ValueError: Invalid graph description, n2 should be >= 0
sage: graphs.BarbellGraph(randint(2, 10^6), randint(-10^6, -1))
Traceback (most recent call last):
...
ValueError: Invalid graph description, n2 should be >= 0
sage: graphs.BarbellGraph(randint(-10^6, 1), randint(-10^6, -1))
Traceback (most recent call last):
...
ValueError: Invalid graph description, n1 should be >= 2
```

static BidiakisCube()

Returns the Bidiakis cube.

For more information, see this Wikipedia article on the 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
True
sage: g.chromatic_number()
3
```

static BiggsSmithGraph(*embedding=1*)

Returns the Biggs-Smith graph.

For more information, see this Wikipedia article on the 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()
```

```
2448
sage: g.show(figsize=[10, 10])    # long time
```

The other embedding:

```
sage: graphs.BiggsSmithGraph(embedding=2).show()
```

TESTS:

```
sage: graphs.BiggsSmithGraph(embedding='xyzzy')
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

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.

INPUTS:

- `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 xrange(3,12):    # long time
....:     H = Graph()
....:     for r in xrange(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()

Returns 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()

Returns 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()

Returns the Brinkmann graph.

For more information, see the Wikipedia article on the 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()

Returns 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.

EXAMPLE:

```
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 $n!$. Thus, the order of $B(n)$ increases according to $f(n) = n!$.

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
```

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
```

TESTS:

Input n must be positive:

```
sage: graphs.BubbleSortGraph(0)
Traceback (most recent call last):
...
ValueError: Invalid number of symbols to permute, n should be >= 1
sage: graphs.BubbleSortGraph(randint(-10^6, 0))
Traceback (most recent call last):
...
ValueError: Invalid number of symbols to permute, n should be >= 1
```

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. This constructor depends on NetworkX numeric labeling. For more information, see this Wikipedia article on the bull graph.

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.

ALGORITHM:

Uses NetworkX.

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. For more information, see this Wikipedia article on the butterfly graph.

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 CameronGraph()

Returns 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 http://www.win.tue.nl/~aeb/graphs/Cameron.html.

EXAMPLES:

```
sage: g = graphs.CameronGraph()
sage: g.order()
231
sage: g.size()
3465
sage: g.is_strongly_regular(parameters = True) # long time
(231, 30, 9, 3)
```

static Cell120()

Returns 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*)

Returns 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 redondant code.

INPUTS:

- 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.

OUTPUTS:

- 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 isomporphic to the cartesian product of 2 complete graphs:

```
sage: G, _ = graphs.ChessboardGraphGenerator( [3,4], rook=True, rook_radius=None, bishop=Fal
sage: H = ( graphs.CompleteGraph(3) ).cartesian_product( graphs.CompleteGraph(4) )
sage: G.is_isomorphic(H)
True
```

TESTS:

Giving dimensions less than 2:

```
sage: graphs.ChessboardGraphGenerator( [0, 2] )
Traceback (most recent call last):
...
ValueError: The dimensions must be positive integers larger than 1.
```

Giving non integer dimensions:

```
sage: graphs.ChessboardGraphGenerator( [4.5, 2] )
Traceback (most recent call last):
...
ValueError: The dimensions must be positive integers larger than 1.
```

Giving too few dimensions:

```
sage: graphs.ChessboardGraphGenerator( [2] )
Traceback (most recent call last):
...
ValueError: The chessboard must have at least 2 dimensions.
```

Giving a non-iterable object as first parameter:

```
sage: graphs.ChessboardGraphGenerator( 2, 3 )
Traceback (most recent call last):
...
TypeError: The first parameter must be an iterable object.
```

Giving too small rook radius:

```
sage: graphs.ChessboardGraphGenerator( [2, 3], rook=True, rook_radius=0 )
Traceback (most recent call last):
...
ValueError: The rook_radius must be either None or have an integer value >= 1.
```

Giving wrong values for knights movements:

```
sage: graphs.ChessboardGraphGenerator( [2, 3], rook=False, bishop=False, knight=True, knight
Traceback (most recent call last):
...
ValueError: The knight_x and knight_y values must be integers of value >= 1.
```

static ChvatalGraph()

Returns 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 this Wikipedia article on the Chvatal 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 adj.

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+3,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 = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(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*)

Returns 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.

This constructor depends on NetworkX numeric labels.

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)th node at the top, then counterclockwise as well.

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 = sage.plot.graphics.GraphicsArray(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.

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*)

Returns a Complete Bipartite Graph sized n1+n2, with each of the nodes [0,(n1-1)] connected to each of the nodes [n1,(n2-1)] and vice versa.

A Complete Bipartite Graph is a graph with its vertices partitioned into two groups, V1 and V2. Each v in V1 is connected to every v in V2, and vice versa.

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,maxn1,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 = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

Trac ticket #12155:

```
sage: graphs.CompleteBipartiteGraph(5,6).complement()
complement(Complete bipartite graph): Graph on 11 vertices
```

static CompleteGraph(*n*)

Returns a complete graph on n nodes.

A Complete Graph is a graph in which all nodes are connected to all other nodes.

This constructor is dependent on vertices numbered 0 through n-1 in NetworkX complete_graph()

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 = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(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
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.

EXAMPLE:

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 CoxeterGraph ()

Return the Coxeter graph.

See the Wikipedia page on the 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 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 = sage.plot.graphics.GraphicsArray(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)

Returns a cycle graph with n nodes.

A cycle graph is a basic structure which is also typically called an n-gon.

This constructor is dependent on vertices numbered 0 through n-1 in NetworkX `cycle_graph()`

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 = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(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 [CharLes1996].

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
```

REFERENCE:

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
```

TESTS:

Trac ticket #12155:

```
sage: graphs.DegreeSequenceBipartite([2,2,2,2,2],[5,5]).complement()
complement(): Graph on 7 vertices
```

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` - for the random number generator.

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: G.edges(labels=False)
[(0, 2), (0, 10), (0, 15), (1, 6), (1, 16), (1, 17), (2, 5), (2, 19), (3, 7), (3, 14), (3, 1
sage: G.show()  # long time
```

REFERENCE:

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` - for the random number generator.

EXAMPLE:

```
sage: G = graphs.DegreeSequenceExpected([1,2,3,2,3])
sage: G.edges(labels=False)
[(0, 2), (0, 3), (1, 1), (1, 4), (2, 3), (2, 4), (3, 4), (4, 4)]
sage: G.show()  # long time
```

REFERENCE:

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.

EXAMPLE:

```
sage: G = graphs.DegreeSequenceTree([3,1,3,3,1,1,1,2,1])
sage: G.show()  # long time
```

static DesarguesGraph()

Returns the Desargues graph.

PLOTTING: The layout chosen is the same as on the cover of [1].

EXAMPLE:

```
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
```

REFERENCE:

- [1] Harary, F. Graph Theory. Reading, MA: Addison-Wesley, 1994.

static DiamondGraph()

Returns a diamond graph with 4 nodes.

A diamond graph is a square with one pair of diagonal nodes connected.

This constructor depends on NetworkX numeric labeling.

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 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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static DorogovtsevGoltsevMendesGraph(*n*)

Construct the n-th generation of the Dorogovtsev-Goltsev-Mendes graph.

EXAMPLE:

```
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()

Returns the double star snark.

The double star snark is a 3-regular graph on 30 vertices. See the Wikipedia page on the 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()

Returns the Dürer graph.

For more information, see this Wikipedia article on the Dürer 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()
Returns the Dyck graph.

For more information, see the MathWorld article on the Dyck graph or the Wikipedia article on the 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
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 EllinghamHorton54Graph()
Returns the Ellingham-Horton 54-graph.

For more information, see the Wikipedia page on the Ellingham-Horton graphs

EXAMPLE:

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
```

TESTS:

```
sage: g.show() # long time
```

static EllinghamHorton78Graph()

Returns the Ellingham-Horton 78-graph.

For more information, see the Wikipedia page on the Ellingham-Horton graphs

EXAMPLE:

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
```

TESTS:

```
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(4)[1:]:
....:     empty2.add_edge(4,i) # add edges {[1:4],[2:4],[3:4]}
sage: empty2.show() # long time
```

static ErreraGraph()

Returns the Errera graph.

For more information, see this Wikipedia article on the 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
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 FibonacciTree(n)

Returns the graph of the Fibonacci Tree F_i of order n. F_i is recursively defined as the a 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()

Returns 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.

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.

REFERENCES:

- [1] Weisstein, E. (1999). "Flower Snark - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/FlowerSnark.html [2007, February 17]

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.

For more information on folded cube graphs, see the corresponding Wikipedia page.

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()

Returns the Folkman graph.

See the Wikipedia page on the Folkman Graph.

EXAMPLE:

```
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()

Returns the Foster graph.

See the Wikipedia page on the Foster Graph.

EXAMPLE:

```
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()

Returns the Franklin graph.

For more information, see this Wikipedia article on the 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)

Returns 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 this Wikipedia article on the 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 = sage.plot.graphics.GraphicsArray(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
```

TESTS:

The input n must be a positive integer.

```
sage: graphs.FriendshipGraph(randint(-10^5, 0))
Traceback (most recent call last):
...
ValueError: n must be a positive integer
```

static FruchtGraph()

Returns 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.

This constructor is dependent on NetworkX's numeric labeling.

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.

REFERENCES:

- [1] Weisstein, E. (1999). "Frucht Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/FruchtGraph.html [2007, February 17]

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 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: graphs.FuzzyBallGraph([3,1],2).adjacency_matrix()
[0 1 1 1 1 1 1 0]
[1 0 1 1 1 1 1 0]
[1 1 0 1 1 1 1 0]
[1 1 1 0 1 1 0 1]
[1 1 1 1 0 1 0 0]
[1 1 1 1 1 0 0 0]
[1 1 1 0 0 0 0 0]
[0 0 0 1 0 0 0 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).charpoly() for g in g_list])  # long time (7s
set([x^8 - 8*x^7 + 4079/150*x^6 - 68689/1350*x^5 + 610783/10800*x^4 - 120877/3240*x^3 + 1351
```

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 Moebius-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 GoldnerHararyGraph()

Return the Goldner-Harary graph.

For more information, see this Wikipedia article on the Goldner-Harary 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 GrayGraph (*embedding=1*)

Returns the Gray graph.

See the Wikipedia page on the 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
```

TESTS:

```
sage: graphs.GrayGraph(embedding = 3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1, 2, or 3.
```

static Grid2dGraph (*n1*, *n2*)

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.

This constructor depends on NetworkX numeric labels.

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
```

TESTS:

Senseless input:

```
sage: graphs.Grid2dGraph(5,0)
Traceback (most recent call last):
...
ValueError: Parameters n1 and n2 must be positive integers !
sage: graphs.Grid2dGraph(-1,0)
```

```
Traceback (most recent call last):
...
ValueError: Parameters n1 and n2 must be positive integers !
```

The graph name contains the dimension:

```
sage: g = graphs.Grid2dGraph(5,7)
sage: g.name()
'2D Grid Graph for [5, 7]'
```

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
```

TESTS:

The graph name contains the dimension:

```
sage: g = graphs.GridGraph([5, 7])
sage: g.name()
'Grid Graph for [5, 7]'
sage: g = graphs.GridGraph([2, 3, 4])
sage: g.name()
'Grid Graph for [2, 3, 4]'
sage: g = graphs.GridGraph([2, 4, 3])
sage: g.name()
'Grid Graph for [2, 4, 3]'
```

All dimensions must be positive integers:

```
sage: g = graphs.GridGraph([2,-1,3])
Traceback (most recent call last):
...
ValueError: All dimensions must be positive integers !
```

static GrotzschGraph()

Returns the Grötzsch graph.

The Grötzsch graph is an example of a triangle-free graph with chromatic number equal to 4. For more information, see this Wikipedia article on Grötzsch graph.

REFERENCE:

- [1] Weisstein, Eric W. "Grotzsch Graph." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GroetzschGraph.html

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 HallJankoGraph (*from_string=True*)

Returns the Hall-Janko graph.

For more information on the Hall-Janko graph, see its Wikipedia page.

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:
```

```
....:         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
```

TESTS:

```
sage: gg = graphs.HallJankoGraph(from_string=False) # long time
sage: g == gg # long time
True
```

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 [ARETT-DOREE].

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
```

TESTS:

It is an error to have just one peg (or less).

```
sage: graphs.HanoiTowerGraph(1, 5)
Traceback (most recent call last):
...
ValueError: Pegs for Tower of Hanoi graph should be two or greater (not 1)
```

It is an error to have zero disks (or less).

```
sage: graphs.HanoiTowerGraph(2, 0)
Traceback (most recent call last):
...
ValueError: Disks for Tower of Hanoi graph should be one or greater (not 0)
```

Citations

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
```

TESTS:

Connectivity of some Harary graphs:

```
sage: n=10
sage: for k in range(2,n):
...       g = graphs.HararyGraph(k,n)
...       if k != g.vertex_connectivity():
...          print "Connectivity of Harary graphs not satisfied."
```

static HarriesGraph(*embedding=1*)

Returns the Harries Graph.

The Harries graph is a Hamiltonian 3-regular graph on 70 vertices. See the Wikipedia page on the 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
```

TESTS:

```
sage: graphs.HarriesGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

static HarriesWongGraph (*embedding=1*)

Returns the Harries-Wong Graph.

See the Wikipedia page on the 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]
sage: g.show(figsize=[15, 15], partition=orbits)    # long time
```

Alternative embedding:

```
sage: graphs.HarriesWongGraph(embedding=2).show()
```

TESTS:

```
sage: graphs.HarriesWongGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

static HeawoodGraph()

Returns a Heawood graph.

The Heawood graph is a cage graph that has 14 nodes. It is a cubic symmetric graph. (See also the Moebius-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.

This constructor is dependent on NetworkX's numeric labeling.

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.

REFERENCES:

- [1] Weisstein, E. (1999). "Heawood Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/HeawoodGraph.html [2007, February 17]

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()

Returns the Herschel graph.

For more information, see this Wikipedia article on the 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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static HigmanSimsGraph(*relabel=True*)

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 [BROUWER-HS-2009]).

Its most famous property is that the automorphism group has an index 2 subgroup which is one of the 26 sporadic groups. [HIGMAN1968]

The construction used here follows [HAFNER2004].

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 [HAFNER2004] 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(range(0,25)+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
```

REFERENCES:

AUTHOR:

- Rob Beezer (2009-10-24)

static HoffmanGraph()
Returns the Hoffman Graph.

See the Wikipedia page on the 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()
Returns 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.

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.

REFERENCES:

EXAMPLES:

```
sage: HS = graphs.HoffmanSingletonGraph()
sage: Set(HS.degree())
{7}
sage: HS.girth()
5
sage: HS.diameter()
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: graphs.HoffmanSingletonGraph().layout()[1]
(-0.904..., 0.425...)
```

static HoltGraph()
Returns the Holt graph (also called the Doyle graph)

See the Wikipedia page on the 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()

Returns 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()
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).

This constructor depends on NetworkX numeric labeling.

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.

This constructor depends on NetworkX numeric labeling.

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
```

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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static IntersectionGraph(*S*)

Returns 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.

EXAMPLE:

```
sage: graphs.IntersectionGraph([(1,2,3),(3,4,5),(5,6,7)])
Intersection Graph: Graph on 3 vertices
```

TESTS:

```
sage: graphs.IntersectionGraph([(1,2,[1])])
Traceback (most recent call last):
...
TypeError: The elements of S must be hashable, and this one is not: (1, 2, [1])
```

static IntervalGraph(*intervals*, *points_ordered=False*)

Returns 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().

EXAMPLE:

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 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. For more information about Johnson graphs, see the corresponding Wikipedia page.

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 Knesser Graph $K(n, 2)$. In paritcular

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.

INPUTS:

- `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 ()

Returns 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 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 $5, 2$.

EXAMPLE:

```
sage: KG=graphs.KneserGraph(5,2)
sage: print KG.vertices()
[{4, 5}, {1, 3}, {2, 5}, {2, 3}, {3, 4}, {3, 5}, {1, 4}, {1, 5}, {1, 2}, {2, 4}]
sage: P=graphs.PetersenGraph()
sage: P.is_isomorphic(KG)
True
```

TESTS:

```
sage: KG=graphs.KneserGraph(0,0)
Traceback (most recent call last):
...
ValueError: Parameter n should be a strictly positive integer
sage: KG=graphs.KneserGraph(5,6)
Traceback (most recent call last):
...
ValueError: Parameter k should be a strictly positive integer inferior to n
```

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 $n > 4$.

INPUTS:

- `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()

Returns a Krackhardt kite graph with 10 nodes.

The Krackhardt kite graph was originally developed by David Krackhardt for the purpose of studying social networks. 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.

REFERENCES:

- [1] Kreps, V. (2002). "Social Network Analysis". [Online] Available: http://www.orgnet.com/sna.html

This constructor depends on NetworkX numeric labeling.

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.

EXAMPLE: Construct and show a Krackhardt kite graph

```
sage: g = graphs.KrackhardtKiteGraph()
sage: g.show() # long time
```

static LCFGraph(*n*, *shift_list*, *repeats*)

Returns 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.

This constructor depends on NetworkX numeric labels.

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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static LjubljanaGraph(*embedding=1*)

Returns 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 page on the 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
```

TESTS:

```
sage: graphs.LjubljanaGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

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).

This constructor depends on NetworkX numeric labels.

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)
sage: g.show() # long time
```

Create several lollipop graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(6):
```

```
....:     k = graphs.LollipopGraph(i+3,4)
....:     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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static M22Graph()

Returns 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 http://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()

Returns 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 McGeeGraph(*embedding=2*)

Returns the McGee Graph.

See the Wikipedia page on the 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()
```

TESTS:

```
sage: graphs.McGeeGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

static McLaughlinGraph()

Returns 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)
sage: set(g.spectrum()) == {112, 2, -28}      # optional gap_packages
True
```

static MeredithGraph()

Returns 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()

Returns a Moebius-Kantor Graph.

A Moebius-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].

PLOTTING: See the plotting section for the generalized Petersen graphs.

REFERENCES:

- [1] Weisstein, E. (1999). "Moebius-Kantor Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/Moebius-KantorGraph.html [2007, February 17]

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()

Returns the Moser spindle.

For more information, see this MathWorld article on the 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 has chromatic number 4 and its automorphism group is isomorphic to the dihedral group D_4.

```
sage: G.chromatic_number()
4
sage: ag = G.automorphism_group()
```

```
sage: ag.is_isomorphic(DihedralGroup(4))
True
```

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.

EXAMPLE:

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
```

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
```

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*)

Returns the Nauru Graph.

See the Wikipedia page on the 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()
```

TESTS:

```
sage: graphs.NauruGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
sage: graphs.NauruGraph(embedding=1).is_isomorphic(g)
True
```

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 = sage.plot.graphics.GraphicsArray(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.

EXAMPLE:

```
sage: OG=graphs.OddGraph(3)
sage: print OG.vertices()
[{4, 5}, {1, 3}, {2, 5}, {2, 3}, {3, 4}, {3, 5}, {1, 4}, {1, 5}, {1, 2}, {2, 4}]
sage: P=graphs.PetersenGraph()
```

```
sage: P.is_isomorphic(OG)
True
```

TESTS:

```
sage: KG=graphs.OddGraph(1)
Traceback (most recent call last):
...
ValueError: Parameter n should be an integer strictly greater than 1
```

static OrthogonalArrayBlockGraph(*k*, *n*, *OA=None*)

Returns 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_array(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],
....:        [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
```

TESTS:

```
sage: G = graphs.OrthogonalArrayBlockGraph(4,6)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build an OA(4,6)!
sage: G = graphs.OrthogonalArrayBlockGraph(8,2)
Traceback (most recent call last):
...
ValueError: There is no OA(8,2). Beware, Brouwer's website uses OA(n,k) instead of OA(k,n) !
```

static `OrthogonalPolarGraph`(*m*, *q*, *sign='+'*)

Returns the Orthogonal Polar Graph $O^\epsilon(m,q)$.

For more information on Orthogonal Polar graphs, see 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)
```

TESTS:

```
sage: G = graphs.OrthogonalPolarGraph(4,3,"")
Traceback (most recent call last):
...
ValueError: sign must be equal to either '-' or '+' when m is even
sage: G = graphs.OrthogonalPolarGraph(5,3,"-")
Traceback (most recent call last):
...
ValueError: sign must be equal to either '' or '+' when m is odd
```

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.complement().is_isomorphic(G)
True
```

static PappusGraph()

Returns 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 PathGraph(n, *pos=None*)

Returns a path graph with n nodes. Pos argument takes a string which is either 'circle' or 'line', (otherwise the default is used). See the plotting section below for more detail.

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).

This constructor depends on NetworkX numeric labels.

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 11

```
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 40

```
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 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.

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),
 (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)]
```

TESTS:

```
sage: graphs.PermutationGraph([1, 2, 3], [4, 5, 6])
Traceback (most recent call last):
...
ValueError: The two permutations do not contain the same set of elements ...
```

static PetersenGraph()

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
sage: petersen_spring.show() # long time
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

static PoussinGraph()

Returns 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 $n \equiv 1, 5 \bmod 6$.

INPUTS:

- `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 xrange(3,12):   # long time
....:     for r in xrange(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())
....:         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` - for random number generator

EXAMPLES:

We show the edge list of a random graph on 6 nodes with m = 2.

```
sage: graphs.RandomBarabasiAlbert(6,2).edges(labels=False)
[(0, 2), (0, 3), (0, 4), (1, 2), (2, 3), (2, 4), (2, 5), (3, 5)]
```

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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show()  # long time
```

static RandomBipartite(*n1*, *n2*, *p*)

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

EXAMPLE:

```
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)]
```

TESTS:

```
sage: g=graphs.RandomBipartite(5,-3,0.5)
Traceback (most recent call last):
...
ValueError: n1 and n2 should be integers strictly greater than 0
sage: g=graphs.RandomBipartite(5,3,1.5)
Traceback (most recent call last):
...
ValueError: Parameter p is a probability, and so should be a real value between 0 and 1
```

Trac ticket #12155:

```
sage: graphs.RandomBipartite(5,6,.2).complement()
complement(Random bipartite graph of size 5+6 with edge probability 0.200000000000000): Grap
```

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.

EXAMPLE:

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 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

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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show()  # long time
```

static RandomGNP(*n*, *p*, *seed=None*, *fast=True*, *method='Sage'*)

Returns a random graph on n nodes. Each edge is inserted independently with probability p.

INPUTS:

- `n` – number of nodes of the graph
- `p` – probability of an edge
- `seed` – integer seed for random number generator (default=None).
- `fast` – boolean set to True (default) to use the algorithm with time complexity in $O(n+m)$ proposed in [BatBra2005]. It is designed for generating large sparse graphs. It is faster than other methods for *LARGE* instances (try it to know whether it is useful for you).
- `method` – By default (`'method='Sage'`), this function uses the method implemented in `'sage.graphs.graph_generators_pyx.pyx`. When `method='networkx'`, this function calls the NetworkX function `fast_gnp_random_graph`, unless `fast=False`, then `gnp_random_graph`. Try them to know which method is the best for you. The `fast` parameter is not taken into account by the 'Sage' method so far.

REFERENCES:

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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
sage: graphs.RandomGNP(4,1)
Complete graph: Graph on 4 vertices
```

TESTS:

```
sage: graphs.RandomGNP(50,.2,method=50)
Traceback (most recent call last):
...
ValueError: 'method' must be equal to 'networkx' or to 'Sage'.
sage: set_random_seed(0)
sage: graphs.RandomGNP(50,.2, method="Sage").size()
243
sage: graphs.RandomGNP(50,.2, method="networkx").size()
258
```

static RandomHolmeKim(*n*, *m*, *p*, *seed=None*)

Returns 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` - for the random number generator.

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.

EXAMPLE: We show the edge list of a random graph on 8 nodes with 2 random edges per node and a probability $p = 0.5$ of forming triangles.

```
sage: graphs.RandomHolmeKim(8, 2, 0.5).edges(labels=False)
[(0, 2), (0, 5), (1, 2), (1, 3), (2, 3), (2, 4), (2, 6), (2, 7),
 (3, 4), (3, 6), (3, 7), (4, 5)]

sage: G = graphs.RandomHolmeKim(12, 3, .3)
sage: G.show()  # long time
```

REFERENCE:

static RandomInterval(*n*)

RandomInterval() is deprecated. Use RandomIntervalGraph() instead.

TEST:

```
sage: g = graphs.RandomInterval(8)
doctest:...: DeprecationWarning: RandomInterval() is deprecated. Use RandomIntervalGraph() i
See http://trac.sagemath.org/13283 for details.
```

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 [boucheron2001].

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.

EXAMPLE:

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
```

REFERENCE:

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 - for the random number generator

EXAMPLE: 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)
[(0, 1), (1, 2)]

sage: G = graphs.RandomLobster(9, .6, .3)
sage: G.show()  # long time
```

static RandomNewmanWattsStrogatz(*n*, *k*, *p*, *seed=None*)

Returns 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 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 - for the random number generator

EXAMPLE: We show the edge list of a random graph on 7 nodes with 2 "nearest neighbors" and probability $p = 0.2$:

```
sage: graphs.RandomNewmanWattsStrogatz(7, 2, 0.2).edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 6), (1, 2), (2, 3), (2, 4), (3, 4), (3, 6), (4, 5), (5, 6)]

sage: G = graphs.RandomNewmanWattsStrogatz(12, 2, .3)
sage: G.show()  # long time
```

REFERENCE:

static RandomRegular(*d*, *n*, *seed=None*)

Returns a random d-regular graph on n vertices, or returns False on failure.

Since every edge is incident to two vertices, n*d must be even.

INPUT:

- n - number of vertices
- d - degree
- seed - for the random number generator

EXAMPLE: We show the edge list of a random graph with 8 nodes each of degree 3.

```
sage: graphs.RandomRegular(3, 8).edges(labels=False)
[(0, 1), (0, 4), (0, 7), (1, 5), (1, 7), (2, 3), (2, 5), (2, 6), (3, 4), (3, 6), (4, 5), (6,

sage: G = graphs.RandomRegular(3, 20)
sage: if G:
....:     G.show()  # random output, long time
```

REFERENCES:

static RandomShell(*constructor*, *seed=None*)

Returns a random shell graph for the constructor given.

INPUT:

- `constructor` - a list of 3-tuples (n,m,d), each representing a shell
- `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` - for the random number generator

EXAMPLE:

```
sage: G = graphs.RandomShell([(10,20,0.8),(20,40,0.8)])
sage: G.edges(labels=False)
[(0, 3), (0, 7), (0, 8), (1, 2), (1, 5), (1, 8), (1, 9), (3, 6), (3, 11), (4, 6), (4, 7), (4
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.

EXAMPLE:

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
```

TEST:

sage: g = graphs.RandomToleranceGraph(-2) Traceback (most recent call last): ... ValueError: The number n of vertices must be >= 0.

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 algoritm 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

EXAMPLE:

```
sage: G = graphs.RandomTree(10)
sage: G.is_tree()
True
sage: G.show() # long
```

TESTS:

Ensuring that we encounter no unexpected surprise

```
sage: all( graphs.RandomTree(10).is_tree()
....:      for i in range(100) )
True
```

static RandomTreePowerlaw(*n*, *gamma=3*, *tries=100*, *seed=None*)

Returns a tree with a power law degree distribution. Returns 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
- tries - number of attempts to adjust sequence to make a tree
- seed - for the random number generator

EXAMPLE: We show the edge list of a random graph with 10 nodes and a power law exponent of 2.

```
sage: graphs.RandomTreePowerlaw(10, 2).edges(labels=False)
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (6, 8), (6, 9)]

sage: G = graphs.RandomTreePowerlaw(15, 2)
sage: if G:
....:     G.show()  # random output, long time
```

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 $v = `2^{i+1} - 1$.

Ringed trees are defined in [CFHM12].

INPUT:

- k – the number of levels of the ringed tree.
- vertex_labels (boolean) – whether to label vertices as binary words (default) or as integers.

EXAMPLE:

```
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']
```

TEST:

```
sage: G = graphs.RingedTree(-1)
Traceback (most recent call last):
...
ValueError: The number of levels must be >= 1.
sage: G = graphs.RingedTree(5, vertex_labels = False)
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, 25, 26, 27, 28, 29, 30]
```

REFERENCES:

static RobertsonGraph()

Returns the Robertson graph.

See the Wikipedia page on the Robertson Graph.

EXAMPLE:

```
sage: g = graphs.RobertsonGraph()
sage: g.order()
19
sage: g.size()
38
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 +
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 $K_n \Box K_m$.

INPUTS:

- `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()

Returns the Schläfli graph.

The Schläfli graph is the only strongly regular graphs of parameters $(27, 16, 10, 8)$ (see [GodsilRoyle]).

For more information, see the Wikipedia article on the 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.

EXAMPLE:

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()

Returns the Shrikhande graph.

For more information, see the MathWorld article on the Shrikhande graph or the Wikipedia article on the 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) ])
set([2])
```

It is non-planar, and both Hamiltonian and Eulerian:

```
sage: G.is_planar()
False
sage: G.is_hamiltonian()
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 SimsGewirtzGraph()

Returns 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 http://www.win.tue.nl/~aeb/graphs/Sims-Gewirtz.html

or its Wikipedia page.

See Also:

- HigmanSimsGraph().

EXAMPLE:

```
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()

Returns the Sousselier Graph.

The Sousselier graph is a hypohamiltonian graph on 16 vertices and 27 edges. For more information, see the corresponding Wikipedia page (in French).

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())
sage: g.is_hamiltonian()
True
```

static StarGraph(*n*)

Returns a star graph with n+1 nodes.

A Star graph is a basic structure where one node is connected to all other nodes.

This constructor is dependent on NetworkX numeric labels.

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):
....:     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 = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static SylvesterGraph()

Returns 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 http://www.win.tue.nl/~aeb/graphs/Sylvester.html.

See Also:

- HoffmanSingletonGraph().

EXAMPLE:

```
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 SymplecticGraph(d, q)

Returns the Symplectic graph $Sp(d, q)$

The Symplectic 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.

EXAMPLES:

```
sage: g = graphs.SymplecticGraph(6,2)
sage: g.is_strongly_regular(parameters=True)
(63, 30, 13, 15)
sage: set(g.spectrum()) == {-5, 3, 30}
True
```

static SzekeresSnarkGraph()

Returns 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 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 = sage.plot.graphics.GraphicsArray(j)
sage: G.show() # long time
```

static ThomsenGraph()

Returns 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()

Returns 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*)

Returns 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 tolerance. 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()`.

EXAMPLE:

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
```

TEST:

Giving negative third value:

```
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)
Traceback (most recent call last):
...
ValueError: Invalid tolerance representation at position 0; third value must be positive!
```

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.

EXAMPLE:

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
```

TESTS:

Senseless input:

```
sage: graphs.Toroidal6RegularGrid2dGraph(5,2)
Traceback (most recent call last):
...
ValueError: Parameters n1 and n2 must be integers larger than 3 !
sage: graphs.Toroidal6RegularGrid2dGraph(2,0)
Traceback (most recent call last):
...
ValueError: Parameters n1 and n2 must be integers larger than 3 !
```

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 $((0, i), (n_1 - 1, i))$.

EXAMPLE:

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 Tutte12Cage()

Returns Tutte's 12-Cage.

See the Wikipedia page on the 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*)

Returns the Tutte-Coxeter graph.

See the Wikipedia page on the 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()
```

TESTS:

```
sage: graphs.TutteCoxeterGraph(embedding=3)
Traceback (most recent call last):
...
ValueError: The value of embedding must be 1 or 2.
```

static TutteGraph()

Returns 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
3
sage: g.girth()
4
sage: g.automorphism_group().cardinality()
3
sage: g.is_hamiltonian()
False
```

static WagnerGraph()

Returns the Wagner Graph.

See the Wikipedia page on the 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()

Returns 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
sage: g.chromatic_number()
3
```

static WellsGraph()

Returns 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 [BCN89]. 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
```

REFERENCES:

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.

This constructor depends on NetworkX numeric labels.

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):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.graphics.GraphicsArray(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 = sage.plot.graphics.GraphicsArray(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()

Returns 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 its (french) Wikipedia page.

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 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.

EXAMPLE:

```
sage: g=graphs.WorldMap()
sage: g.has_edge("France","Italy")
True
sage: g.gps_coordinates["Bolivia"]
[[17, 'S'], [65, 'W']]
sage: sorted(g.connected_component_containing_vertex('Ireland'))
['Ireland', 'United Kingdom']
```

REFERENCE:

cospectral_graphs(*vertices*, *matrix_function=<function <lambda> at 0x7fb30a1f8ed8>*, *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 cadinality 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)
...       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
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(normali
True
```

fullerenes(*order*, *ipr=False*)

Returns a generator which creates fullerene graphs using the buckygen generator (see [buckygen]).

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: gen.next()  # optional buckygen
Graph on 60 vertices
sage: gen.next()  # 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 = gen.next()  # 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],
 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
```

REFERENCE:

fusenes(*hexagon_count*, *benzenoids=False*)

Returns a generator which creates fusenes and benzenoids using the benzene generator (see [benzene]). 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 naphtalene ($C_{10}H_8$). 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: gen.next()  # optional benzene
Graph on 10 vertices
sage: gen.next()  # optional benzene
Traceback (most recent call last):
...
StopIteration
```

There are 6505 benzenoids with 9 hexagons:

```
sage: gen = graphs.fusenes(9, benzenoids=True)  # optional benzene
sage: len(list(gen))  # optional benzene
6505
```

REFERENCE:

static line_graph_forbidden_subgraphs()

Returns the 9 forbidden subgraphs of a line graph.

Wikipedia article on the line graphs

The graphs are returned in the ordering given by the Wikipedia drawing, read from left to right and from top to bottom.

EXAMPLE:

```
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*)

Returns a generator which creates graphs from nauty's geng program.

Note: Due to license restrictions, the nauty package is distributed as a Sage optional package. At a system command line, execute `sage -i nauty` to see the nauty license and install the package.

INPUT:

- `options` - 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` - 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
                 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") # optional nauty
sage: gen.next() # optional nauty
Graph on 2 vertices
sage: gen.next() # optional nauty
Graph on 2 vertices
sage: gen.next() # optional nauty
Traceback (most recent call last):
...
StopIteration: Exhausted list of graphs from nauty geng
```

A list of all graphs on 7 vertices. This agrees with Sloane's OEIS sequence A000088.

```
sage: gen = graphs.nauty_geng("7") # optional nauty
sage: len(list(gen))  # optional nauty
1044
```

A list of just the connected graphs on 7 vertices. This agrees with Sloane's OEIS sequence A001349.

```
sage: gen = graphs.nauty_geng("7 -c") # optional nauty
sage: len(list(gen))  # optional nauty
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.

```
sage: gen = graphs.nauty_geng("4", debug=True) # optional nauty
sage: print gen.next() # optional nauty
>A nauty-geng -d0D3 n=4 e=0-6
```

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.

EXAMPLE:

```
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
```

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]
```

`sage.graphs.graph_generators.`**canaug_traverse_edge**(*g*, *aut_gens*, *property*, *dig=False*, *loops=False*, *implementation='c_graph'*, *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*, *implementation='c_graph'*, *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.

EXAMPLE: 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.

EXAMPLE: 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()`	Returns a n-dimensional butterfly graph.
`Circuit()`	Returns the circuit on n vertices.
`Circulant()`	Returns a circulant digraph on n vertices from a set of integers.
`DeBruijn()`	Returns the De Bruijn digraph with parameters k, n.
`GeneralizedDeBruijn()`	Returns the generalized de Bruijn digraph of order n and degree d.
`ImaseItoh()`	Returns the digraph of Imase and Itoh of order n and degree d.
`Kautz()`	Returns the Kautz digraph of degree d and diameter D.
`Path()`	Returns a directed path on n vertices.
`RandomDirectedGNC()`	Returns a random GNC (growing network with copying) digraph with n vertices.
`RandomDirectedGNM()`	Returns a random labelled digraph on n nodes and m arcs.
`RandomDirectedGNP()`	Returns a random digraph on n nodes.
`RandomDirectedGN()`	Returns a random GN (growing network) digraph with n vertices.
`RandomDirectedGNR()`	Returns a random GNR (growing network with redirection) digraph.
`RandomTournament()`	Returns a random tournament on n vertices.
`TransitiveTournament()`	Returns a transitive tournament on n vertices.
`tournaments_nauty()`	Returns 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`**

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

Families of Graphs:
    - 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. Iterates over distinct, exhaustive representatives.

INPUT:

- `vertices` - natural number
- `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` - ignored if implementation is not `c_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
```

Note that we can also get digraphs with underlying Cython implementation:

```
sage: for D in digraphs(2, augment='vertices', implementation='c_graph'):
...     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(0, 5):
...     print len(list(digraphs(i)))
1
1
3
16
218
```

REFERENCE:

- Brendan D. McKay, Isomorph-Free Exhaustive generation. Journal of Algorithms Volume 26, Issue 2, February 1998, pages 306-324.

ButterflyGraph (*n*, *vertices='strings'*)

Returns 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 v[i] != w[i].

A butterfly graph has $(2^n)(n+1)$ vertices and $n2^{n+1}$ edges.

INPUT:

- `vertices` - 'strings' (default) or 'vectors', specifying whether the vertices are zero-one strings or actually tuples over GF(2).

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*)

Returns the circuit on n vertices

The circuit is an oriented `CycleGraph`

EXAMPLE:

A circuit is the smallest strongly connected digraph:

```
sage: circuit = digraphs.Circuit(15)
sage: len(circuit.strongly_connected_components()) == 1
True
```

Circulant(*n*, *integers*)

Returns a circulant digraph on n vertices from a set of integers.

INPUT:

- `n` (integer) – number of vertices.
- `integers` – the list of integers such that there is an edge from i to j if and only if `(j-i)%n in integers`.

EXAMPLE:

```
sage: digraphs.Circulant(13,[3,5,7])
Circulant graph ([3, 5, 7]): Digraph on 13 vertices
```

TESTS:

```
sage: digraphs.Circulant(13,[3,5,7,"hey"])
Traceback (most recent call last):
...
ValueError: The list must contain only relative integers.
sage: digraphs.Circulant(3,[3,5,7,3.4])
Traceback (most recent call last):
```

```
...
ValueError: The list must contain only relative integers.
```

DeBruijn(*k*, *n*, *vertices='strings'*)

Returns 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 on 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` – An integer equal to the length of words in the De Bruijn digraph when `vertices == 'strings'`, and also to the diameter of the digraph.
- `vertices` – 'strings' (default) or 'integers', specifying whether the vertices are words build upon an alphabet or integers.

EXAMPLES:

```
sage: db=digraphs.DeBruijn(2,2); db
De Bruijn digraph (k=2, n=2): Looped digraph on 4 vertices
sage: db.order()
4
sage: db.size()
8
```

TESTS:

```
sage: digraphs.DeBruijn(5,0)
De Bruijn digraph (k=5, n=0): Looped multi-digraph on 1 vertex
sage: digraphs.DeBruijn(0,0)
De Bruijn digraph (k=0, n=0): Looped multi-digraph on 0 vertices
```

GeneralizedDeBruijn(*n*, *d*)

Returns the generalized de Bruijn digraph of order n and degree d.

The generalized de Bruijn digraph has been defined in [RPK80] [RPK83]. 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.

INPUTS:

- `n` – is the number of vertices of the digraph
- `d` – is the degree of the digraph

See Also:

- sage.graphs.generic_graph.GenericGraph.is_circulant() – checks whether a (di)graph is circulant, and/or returns all possible sets of parameters.

EXAMPLE:

```
sage: GB = digraphs.GeneralizedDeBruijn(8, 2)
sage: GB.is_isomorphic(digraphs.DeBruijn(2, 3), certify = True)
(True, {0: '000', 1: '001', 2: '010', 3: '011', 4: '100', 5: '101', 6: '110', 7: '111'})
```

TESTS:

An exception is raised when the degree is less than one:

```
sage: G = digraphs.GeneralizedDeBruijn(2, 0)
Traceback (most recent call last):
...
ValueError: The generalized de Bruijn digraph is defined for degree at least one.
```

An exception is raised when the order of the graph is less than one:

```
sage: G = digraphs.GeneralizedDeBruijn(0, 2)
Traceback (most recent call last):
...
ValueError: The generalized de Bruijn digraph is defined for at least one vertex.
```

REFERENCES:

ImaseItoh(n, d)

Returns the digraph of Imase and Itoh of order n and degree d.

The digraph of Imase and Itoh has been defined in [II83]. 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 digraph of Imase and Itoh is isomorphic to the de Bruijn digraph of degree d and diameter D. When $n = d^{D-1}(d+1)$, the digraph of Imase and Itoh is isomorphic to the Kautz digraph [Kautz68] of degree d and diameter D.

INPUTS:

- n – is the number of vertices of the digraph
- d – is the degree of the digraph

EXAMPLES:

```
sage: II = digraphs.ImaseItoh(8, 2)
sage: II.is_isomorphic(digraphs.DeBruijn(2, 3), certify = 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: II.is_isomorphic(digraphs.Kautz(2, 3), certify = True)
(True, {0: '010', 1: '012', 2: '021', 3: '020', 4: '202', 5: '201', 6: '210', 7: '212', 8: '
```

TESTS:

An exception is raised when the degree is less than one:

```
sage: G = digraphs.ImaseItoh(2, 0)
Traceback (most recent call last):
...
ValueError: The digraph of Imase and Itoh is defined for degree at least one.
```

An exception is raised when the order of the graph is less than two:

```
sage: G = digraphs.ImaseItoh(1, 2)
Traceback (most recent call last):
...
ValueError: The digraph of Imase and Itoh is defined for at least two vertices.
```

REFERENCE:

Kautz (*k*, *D*, *vertices='strings'*)

Returns the Kautz digraph of degree d and diameter D.

The Kautz digraph has been defined in [Kautz68]. The Kautz digraph of degree d and diameter D has $d^{D-1}(d+1)$ vertices. This digraph is build upon a set of vertices equal to the set of words of length D from an alphabet of $d+1$ letters such that consecutive letters are differents. 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 digraph of Imase and Itoh [II83] of degree d and order $d^{D-1}(d+1)$.

See also the Wikipedia article on Kautz Graphs.

INPUTS:

- **k – Two possibilities for this parameter :**
  - An integer equal to the degree of the digraph to be produced, that is the cardinality minus one of the alphabet to use.
  - 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 – An integer equal to the diameter of the digraph, and also to** the length of a vertex label when `vertices == 'strings'`.
- **vertices – 'strings' (default) or 'integers', specifying whether** the vertices are words build upon an alphabet or integers.

EXAMPLES:

```
sage: K = digraphs.Kautz(2, 3)
sage: K.is_isomorphic(digraphs.ImaseItoh(12, 2), certify = True)
(True, {'201': 5, '120': 9, '202': 4, '212': 7, '210': 6, '010': 0, '121': 8, '012': 1, '021

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

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', 'B
```

TESTS:

An exception is raised when the degree is less than one:

```
sage: G = digraphs.Kautz(0, 2)
Traceback (most recent call last):
...
ValueError: Kautz digraphs are defined for degree at least one.

sage: G = digraphs.Kautz(['a'], 2)
Traceback (most recent call last):
```

```
...
ValueError: Kautz digraphs are defined for degree at least one.
```

An exception is raised when the diameter of the graph is less than one:

```
sage: G = digraphs.Kautz(2, 0)
Traceback (most recent call last):
...
ValueError: Kautz digraphs are defined for diameter at least one.
```

REFERENCE:

Path(*n*)

Returns 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 <lambda> at 0x7fb3108fa410>*, *seed=None*)

Returns a random GN (growing network) 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.

INPUT:

- n - number of vertices.
- kernel - the attachment kernel
- seed - for the random number generator

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGN(25)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (3, 1), (4, 0), (5, 0), (6, 1), (7, 0), (8, 3), (9, 0), (10, 8), (11, 3), (
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Organization of Growing Random Networks, Phys. Rev. E vol. 63 (2001), p. 066123.

RandomDirectedGNC(*n*, *seed=None*)

Returns a random GNC (growing network with copying) 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.

INPUT:

- `n` - number of vertices.
- `seed` - for the random number generator

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGNC(25)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (2, 1), (3, 0), (4, 0), (4, 1), (5, 0), (5, 1), (5, 2), (6, 0), (6, 1), (7,
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Network Growth by Copying, Phys. Rev. E vol. 71 (2005), p. 036118.

RandomDirectedGNM(*n*, *m*, *loops=False*)

Returns a random labelled digraph on n nodes and m arcs.

INPUT:

- `n` (integer) – number of vertices.
- `m` (integer) – number of edges.
- `loops` (boolean) – whether to allow loops (set to `False` by default).

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLE:

```
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,
```

TESTS:

```
sage: digraphs.RandomDirectedGNM(10,-3)
Traceback (most recent call last):
...
ValueError: The number of edges must satisfy 0<= m <= n(n-1) when no loops are allowed, and

sage: digraphs.RandomDirectedGNM(10,100)
Traceback (most recent call last):
...
ValueError: The number of edges must satisfy 0<= m <= n(n-1) when no loops are allowed, and
```

RandomDirectedGNP(*n*, *p*, *loops=False*, *seed=None*)

Returns a random digraph on n nodes. Each edge is inserted independently with probability p.

INPUTS:

- n – number of nodes of the digraph
- p – probability of an edge
- loops – is a boolean set to True if the random digraph may have loops, and False (default) otherwise.
- seed – integer seed for random number generator (default=None).

REFERENCES:

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLE:

```
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,
```

RandomDirectedGNR(*n*, *p*, *seed=None*)

Returns a random GNR (growing network with redirection) 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.

INPUT:

- n - number of vertices.
- p - redirection probability
- seed - for the random number generator.

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGNR(25, .2)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (2, 1), (3, 0), (4, 0), (4, 1), (5, 0), (5, 1), (5, 2), (6, 0), (6, 1), (7,
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Organization of Growing Random Networks, Phys. Rev. E vol. 63 (2001), p. 066123.

RandomTournament(*n*)

Returns 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.

See Wikipedia article Tournament_(graph_theory)

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: digraphs.RandomTournament(-1)
Traceback (most recent call last):
...
ValueError: The number of vertices cannot be strictly negative!
```

TransitiveTournament(*n*)

Returns a transitive tournament on n vertices.

In this tournament there is an edge from i to j if $i < j$.

See Wikipedia article Tournament_(graph_theory)

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
```

TESTS:

```
sage: digraphs.TransitiveTournament(-1)
Traceback (most recent call last):
...
ValueError: The number of vertices cannot be strictly negative!
```

tournaments_nauty(*n*, *min_out_degree=None*, *max_out_degree=None*, *strongly_connected=False*, *debug=False*, *options=''*)

Returns 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) – 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/.

Note: To use this method you must first install the Nauty spkg.

EXAMPLES:

```
sage: for g in digraphs.tournaments_nauty(4): # optional - nauty
....:     print g.edges(labels = False)        # optional - nauty
[(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)] # optional - nauty
[1, 1, 2, 4, 12, 56, 456]
sage: [len(list(tournaments(x, strongly_connected = True))) for x in range(1,9)] # optional
[1, 0, 1, 1, 6, 35, 353, 6008]
```

2.3 Common graphs and digraphs generators (Cython)

Common graphs and digraphs generators (Cython)

AUTHORS:

- David Coudert (2012)

sage.graphs.graph_generators_pyx.**RandomGNP**(*n*, *p*, *directed=False*, *loops=False*)

Returns a random graph or a digraph on n nodes. Each edge is inserted independently with probability p.

INPUTS:

- `n` – number of nodes of the digraph
- `p` – probability of an edge
- `directed` – is a boolean indicating whether the random graph is directed or undirected (default).
- `loops` – is a boolean set to True if the random digraph may have loops, and False (default) otherwise. This value is used only when `directed == True`.

REFERENCES:

EXAMPLE:

```
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),
```

TESTS:

```
sage: abs(mean([RandomGNP(200,.2).density() for i in range(30)])-.2) < .001
True
sage: RandomGNP(150,.2, loops = True)
Traceback (most recent call last):
...
ValueError: The 'loops' argument can be set to True only when 'directed' is True.
```

2.4 Graph database

INFO:

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.database.py).

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_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.database.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:

- database - the GraphDatabase instance to query (if None then a new instance is created)
- query_string - a string representing the SQL query
- param_tuple - a tuple of strings - 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.

See GraphDatabase class docstrings or enter:

```
sage: G = GraphDatabase()
sage: G.get_skeleton()
{...
```

to see the underlying structure of the database. Also see SQLQuery in sage.databases.database 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
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

INFO:

This class interfaces with the sqlite database graphs.db. It is an immutable database that inherits from SQLDatabase (see sage.databases.database.py). 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.

USE: 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 1. 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. 2. GraphQuery - is 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/

EXAMPLE:

```
sage: G = GraphDatabase()
sage: G.get_skeleton()
{u'aut_grp':    {u'edge_transitive':        {'index': True,
                                             'unique': False,
                                             'primary_key': False,
                                             'sql': u'BOOLEAN'},
                 u'vertex_transitive':      {'index': True,
                                             'unique': False,
                                             'primary_key': False,
                                             'sql': u'BOOLEAN'},
                 u'aut_grp_size':           {'index': True,
                                             'unique': False,
                                             'primary_key': False,
                                             'sql': u'INTEGER'},
                 u'graph_id':               {'index': False,
                                             'unique': False,
                                             'primary_key': False,
                                             'sql': u'INTEGER'},
                 u'num_orbits':             {'index': True,
```

```
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'},
                    u'num_fixed_points': {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'}},
 u'degrees':    {u'graph_id':           {'index': False,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'},
                    u'degrees_sd':      {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'REAL'},
                    u'max_degree':      {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'},
                    u'regular':         {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'BOOLEAN'},
                    u'average_degree':  {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'REAL'},
                    u'degree_sequence': {'index': False,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'},
                    u'min_degree':      {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'}},
 u'spectrum':   {u'max_eigenvalue':     {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'REAL'},
                    u'energy':          {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'REAL'},
                    u'spectrum':        {'index': False,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'TEXT'},
                    u'eigenvalues_sd':  {'index': True,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'REAL'},
                    u'graph_id':        {'index': False,
                                        'unique': False,
                                        'primary_key': False,
                                        'sql': u'INTEGER'},
                    u'min_eigenvalue':  {'index': True,
                                        'unique': False,
                                        'primary_key': False,
```

```
                                          'sql': u'REAL'}},
 u'misc':       {u'diameter':             {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'vertex_connectivity':  {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'BOOLEAN'},
                 u'graph_id':             {'index': False,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'num_components':       {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'min_vertex_cover_size': {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'edge_connectivity':    {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'BOOLEAN'},
                 u'num_spanning_trees':   {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'induced_subgraphs':    {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'TEXT'},
                 u'radius':               {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'num_cut_vertices':     {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'clique_number':        {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'independence_number':  {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'},
                 u'girth':                {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'INTEGER'}},
 u'graph_data': {u'perfect':              {'index': True,
                                           'unique': False,
                                           'primary_key': False,
                                           'sql': u'BOOLEAN'},
                 u'planar':               {'index': True,
```

```
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'BOOLEAN'},
                u'graph_id':    {'index': True,
                                 'unique': True,
                                 'primary_key': False,
                                 'sql': u'INTEGER'},
                u'complement_graph6':  {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'TEXT'},
                u'num_edges':   {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'INTEGER'},
                u'num_cycles':  {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'INTEGER'},
                u'graph6':      {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'TEXT'},
                u'num_hamiltonian_cycles': {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'INTEGER'},
                u'lovasz_number':  {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'REAL'},
                u'eulerian':    {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'BOOLEAN'},
                u'num_vertices':  {'index': True,
                                 'unique': False,
                                 'primary_key': False,
                                 'sql': u'INTEGER'}}}
```

interactive_query(*display_cols*, ***kwds*)

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.

Generates an interact shell (in the notebook only) that allows the user to manipulate query parameters and see the updated results.

EXAMPLE:

```
sage: D = GraphDatabase()
sage: D.interactive_query(display_cols=['graph6','num_vertices','degree_sequence'],num_edges
<html>...</html>
```

query(*query_dict=None*, *display_cols=None*, ***kwds*)

Creates a GraphQuery on this database. For full class details, type GraphQuery? and press shift+enter.

EXAMPLE:

```
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]
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]
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 SQLQuery class located in sage.databases.database.py 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: SQLQuery functions are available for GraphQuery. See sage.dataabases.database.py for more details.

INPUT:

- `graph_db` - The GraphDatabase instance to apply the query to. (If None, then a new instance is created).
- `query_dict` - 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` - 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: 1. ['one_of',String,...,String] Will search for graphs containing a subgraph isomorphic to any of the graph6 strings in the list. 2. ['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]
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()

Returns a list of Sage Graph objects that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6','num_vertices','degree_sequence'],num_edges=['<=
sage: L = Q.get_graphs_list()
sage: L[0]
Graph on 2 vertices
sage: len(L)
35
```

number_of()

Returns the number of graphs in the database that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6','num_vertices','degree_sequence'],num_edges=['<=
sage: Q.number_of()
35
```

query_iterator()

Returns an iterator over the results list of the GraphQuery.

EXAMPLE:

```
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 it.next().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*)

Displays the results of a query in table format.

INPUT:

- `max_field_size` - width of fields in command prompt version
- `with_picture` - 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=
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_vertic
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_deg
degree_sequence
--------------------
211
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*)

Takes the database integer data type (one digit per vertex representing its degree, sorted high to low) and converts it to degree sequence list. The graph6 identifier is required for all graphs with no edges, so that the correct number of zeros will be returned.

EXAMPLE:

```
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*)

Takes a degree sequence list (of Integers) and converts to a sorted (max-min) integer data type, as used for faster access in the underlying database.

EXAMPLE:

```
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*)

Constructs a graph from a graph6 string and returns a Graphics object with arguments preset for show function.

EXAMPLE:

```
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*)

Returns a dictionary of allowed table and column names.

INPUT:

- `tablename` - restricts the output to a single table

EXAMPLE:

```
sage: graph_db_info().keys()
['graph_data', 'degrees', 'spectrum', 'misc', 'aut_grp']

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*)

Constructs and returns a GraphQuery object respecting the special input required for the induced_subgraphs parameter. This input can be an individual graph6 string (in which case it is evaluated without the use of this method) or a list of strings. In the latter case, the list should be of one of the following two formats: 1. ['one_of',String,...,String] Will search for graphs containing a subgraph isomorphic to any of the graph6 strings in the list. 2. ['all_of',String,...,String] Will search for graphs containing a subgraph isomorphic to each of the graph6 strings in the list.

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.

EXAMPLE:

```
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 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.

2.5.1 How to use it?

The current limited aim of this module is to provide a pure Sage/Python interface which is easier to deal with than a XML file. I hope it will be rewritten many times until it stabilizes :-)

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
```

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.

Warning: 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
```

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
-------------------------
type                          :  base
ID                            :  gc_32
name                          :  chordal

Problems :
-----------
3-Colourability               :  Linear
Clique                        :  Polynomial
Clique cover                  :  Polynomial
Cliquewidth                   :  Unbounded
Cliquewidth expression        :  NP-complete
Colourability                 :  Linear
Domination                    :  NP-complete
Independent set               :  Linear
```

```
Recognition                    :  Linear
Treewidth                      :  Polynomial
Weighted clique                :  Polynomial
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
```

2.5.2 Predefined classes

`graph_classes` currently predefines the following graph classes

Class	Related methods
BinaryTrees	`BalancedTree()`, `is_tree()`
Bipartite	`BalancedTree()`, `is_bipartite()`
Block	`blocks_and_cut_vertices()`,
Chordal	`is_chordal()`
Comparability	
Gallai	`is_gallai_tree()`
Grid	`Grid2dGraph()`, `GridGraph()`
Interval	`RandomInterval()`, `IntervalGraph()`, `is_interval()`
Line	`line_graph_forbidden_subgraphs()`, `is_line_graph()`
Modular	`modular_decomposition()`
Outerplanar	`is_circular_planar()`
Perfect	`is_perfect()`
Planar	`is_planar()`
Split	`is_split()`
Tree	`trees()`, `is_tree()`
UnitDisk	`IntervalGraph()`,
UnitInterval	`is_interval()`

2.5.3 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
{'problems':
    {'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
```

2.5.4 Information for developpers

- 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`
- 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.

2.5.5 AUTHORS:

- H.N. de Ridder et al. (ISGCI database)
- Nathann Cohen (Sage implementation)

2.5.6 Methods

class sage.graphs.isgci.**GraphClass**(*name*, *gc_id*)

Bases: sage.structure.sage_object.SageObject, sage.structure.unique_representation.CachedR

An instance of this class represents a Graph Class, matching some entry in the ISGCI database.

EXAMPLE:

Testing the inclusion of two classes:

```
sage: Chordal = graph_classes.Chordal
sage: Trees = graph_classes.Tree
sage: Trees <= Chordal
True
sage: Chordal <= Trees
Unknown
```

TEST:

```
sage: Trees >= Chordal
Unknown
sage: Chordal >= Trees
True
```

description()

Prints the information of ISGCI about the current class.

EXAMPLE:

```
sage: graph_classes.Chordal.description()
Class of graphs : Chordal
-------------------------
type                          :  base
ID                            :  gc_32
name                          :  chordal

Problems :
-----------
3-Colourability               :  Linear
Clique                        :  Polynomial
Clique cover                  :  Polynomial
Cliquewidth                   :  Unbounded
Cliquewidth expression        :  NP-complete
Colourability                 :  Linear
Domination                    :  NP-complete
Independent set               :  Linear
Recognition                   :  Linear
Treewidth                     :  Polynomial
Weighted clique               :  Polynomial
Weighted independent set      :  Linear
```

class sage.graphs.isgci.**GraphClasses**

Bases: sage.structure.unique_representation.UniqueRepresentation

x.__init__(...) initializes x; see help(type(x)) for signature

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)
<type 'dict'>
sage: sorted(t["gc_151"].keys())
['ID', 'name', 'problems', 'type']
sage: t["gc_151"]['name']
'cograph'
sage: t["gc_151"]['problems']['Clique']
'Linear'
```

get_class(*id*)

Returns the class corresponding to the given id in the ISGCI database.

INPUT:

- `id` (string) – the desired class' ID

EXAMPLE:

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
```

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)
<type 'list'>
```

```
sage: t[0]
{'super': 'gc_2', 'sub': 'gc_1'}
```

show_all()

Prints all graph classes stored in ISGCI

EXAMPLE:

```
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                  |
...
```

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).

EXAMPLE:

```
sage: graph_classes.update_db() # Not tested -- requires internet
```

CHAPTER

THREE

LOW-LEVEL IMPLEMENTATION

3.1 Fast compiled graphs

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.

3.1.1 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 the given arc to this graph.

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:

- add_arc – add_arc method for sparse graphs.
- add_arc – add_arc method for dense graphs.

EXAMPLE:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.add_arc(0, 1)
Traceback (most recent call last):
...
NotImplementedError
```

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()
[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]
```

TESTS:

```
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(3, extra_vertices=0)
sage: G.add_vertex(6)
Traceback (most recent call last):
...
RuntimeError: Requested vertex is past twice the allocated range: use realloc.

sage: from sage.graphs.base.dense_graph import DenseGraph
sage: G = DenseGraph(3, extra_vertices=0)
sage: G.add_vertex(6)
Traceback (most recent call last):
...
RuntimeError: Requested vertex is past twice the allocated range: use realloc.
```

add_vertices(*verts*)

Adds 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.

EXAMPLE:

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.

EXAMPLE:

```
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*)

Checks 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)
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.

INPUT:

- None.

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
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.

OUTPUT:

•Raise NotImplementedError. This method is not implemented at the CGraph level. A child class should provide a suitable implementation.

See Also:

•del_all_arcs – del_all_arcs method for sparse graphs.

•del_all_arcs – del_all_arcs method for dense graphs.

EXAMPLE:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.del_all_arcs(0,1)
Traceback (most recent call last):
...
NotImplementedError
```

del_vertex(*v*)

Deletes 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.

OUTPUT:

•None.

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 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 xrange(3):
...       for j in xrange(3):
...           if G.has_arc(i, j):
...               print 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 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*)

Determine whether or not the given arc 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.

See Also:

- `has_arc` – has_arc method for sparse graphs.
- `has_arc` – has_arc method for dense graphs.

EXAMPLE:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.has_arc(0, 1)
Not Implemented!
False
```

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*)

Gives the 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.

See Also:

- in_neighbors – in_neighbors method for sparse graphs.
- in_neighbors – in_neighbors method for dense graphs.

EXAMPLE:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.in_neighbors(0)
Traceback (most recent call last):
...
NotImplementedError
```

out_neighbors(*u*)

Gives the out-neighbors of the vertex u.

INPUT:

- u – 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.

See Also:

- out_neighbors – out_neighbors implementation for sparse graphs.
- out_neighbors – out_neighbors implementation for dense graphs.

EXAMPLE:

```
sage: from sage.graphs.base.c_graph import CGraph
sage: G = CGraph()
sage: G.out_neighbors(0)
Traceback (most recent call last):
...
NotImplementedError
```

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)
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()

Returns a list of the vertices in `self`.

INPUT:

- None.

OUTPUT:

- A list of all vertices in this graph.

EXAMPLE:

```
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):
...
AttributeError: 'CGraphBackend' object has no attribute 'vertex_ints'
```

The appropriate way to use these backends is via Sage graphs:

```
sage: G = Graph(30, implementation="c_graph")
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)]
```

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.

EXAMPLE:

```
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.

EXAMPLE:

```
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*)

Returns the shortest path between `x` and `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`.

OUTPUT:

- A list of vertices in the shortest path from `x` to `y`.

EXAMPLE:

```
sage: G = Graph(graphs.PetersenGraph(), implementation="c_graph")
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]
```

TEST:

Bugfix from #7673

```
sage: G = Graph(implementation="networkx")
sage: G.add_edges([(0,1,9),(0,2,8),(1,2,7)])
sage: Gc = G.copy(implementation='c_graph')
sage: sp = G.shortest_path_length(0,1,by_weight=True)
sage: spc = Gc.shortest_path_length(0,1,by_weight=True)
sage: sp == spc
True
```

breadth_first_search(*v*, *reverse=False*, *ignore_direction=False*)

Returns 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
   (a) $v \leftarrow$ dequeue(Q)
   (b) for each $w \in$ adj(v) do # for digraphs, use out-neighbor set oadj(v)
      i. if $D[w] = \infty$ then
         A. $D[w] \leftarrow D[v] + 1$
         B. enqueue(Q, w)
         C. 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(), implementation="c_graph")
sage: list(G.breadth_first_search(0))
[0, 1, 4, 5, 2, 6, 3, 9, 7, 8]
```

Visiting German cities using breadth-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"]}, implementation="c_graph")
sage: list(G.breadth_first_search("Frankfurt"))
['Frankfurt', 'Mannheim', 'Kassel', 'Wurzburg', 'Karlsruhe', 'Munchen', 'Erfurt', 'Nurnberg'
```

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`.

EXAMPLE:

```
sage: from sage.graphs.base.sparse_graph import SparseGraphBackend
sage: B = SparseGraphBackend(7)
sage: B.degree(3, False)
0
```

TESTS:

Ensure that ticket trac ticket #8395 is fixed.

```
sage: def my_add_edges(G, m, n):
...       for i in range(m):
...           u = randint(0, n)
...           v = randint(0, n)
...           G.add_edge(u, v)
sage: G = Graph({1:[1]}); G
Looped graph on 1 vertex
sage: G.edges(labels=False)
[(1, 1)]
sage: G.degree(); G.size()
[2]
1
sage: sum(G.degree()) == 2 * G.size()
True
sage: G = Graph({1:[1,2], 2:[2,3]}, loops=True); G
Looped graph on 3 vertices
sage: my_add_edges(G, 100, 50)
sage: sum(G.degree()) == 2 * G.size()
True
sage: G = Graph({1:[2,2], 2:[3]}); G
Multi-graph on 3 vertices
sage: G.edges(labels=False)
[(1, 2), (1, 2), (2, 3)]
sage: G.degree(); G.size()
[2, 3, 1]
3
sage: sum(G.degree()) == 2 * G.size()
True
sage: G.allow_loops(True); G
Looped multi-graph on 3 vertices
```

```
sage: my_add_edges(G, 100, 50)
sage: sum(G.degree()) == 2 * G.size()
True
sage: D = DiGraph({1:[2], 2:[1,3]}); D
Digraph on 3 vertices
sage: D.edges(labels=False)
[(1, 2), (2, 1), (2, 3)]
sage: D.degree(); D.size()
[2, 3, 1]
3
sage: sum(D.degree()) == 2 * D.size()
True
sage: D.allow_loops(True); D
Looped digraph on 3 vertices
sage: my_add_edges(D, 100, 50)
sage: sum(D.degree()) == 2 * D.size()
True
sage: D.allow_multiple_edges(True)
sage: my_add_edges(D, 200, 50)
sage: sum(D.degree()) == 2 * D.size()
True
sage: G = Graph({1:[2,2,2]})
sage: G.allow_loops(True)
sage: G.add_edge(1,1)
sage: G.add_edge(1,1)
sage: G.edges(labels=False)
[(1, 1), (1, 1), (1, 2), (1, 2), (1, 2)]
sage: G.degree(1)
7
sage: G.allow_loops(False)
sage: G.edges(labels=False)
[(1, 2), (1, 2), (1, 2)]
sage: G.degree(1)
3
sage: G = Graph({1:{2:['a','a','a']}})
sage: G.allow_loops(True)
sage: G.add_edge(1,1,'b')
sage: G.add_edge(1,1,'b')
sage: G.add_edge(1,1)
sage: G.add_edge(1,1)
sage: G.edges()
[(1, 1, None), (1, 1, None), (1, 1, 'b'), (1, 1, 'b'), (1, 2, 'a'), (1, 2, 'a'), (1, 2, 'a')
sage: G.degree(1)
11
sage: G.allow_loops(False)
sage: G.edges()
[(1, 2, 'a'), (1, 2, 'a'), (1, 2, 'a')]
sage: G.degree(1)
3
sage: G = Graph({1:{2:['a','a','a']}})
sage: G.allow_loops(True)
sage: G.add_edge(1,1,'b')
sage: G.add_edge(1,1,'b')
sage: G.edges()
[(1, 1, 'b'), (1, 1, 'b'), (1, 2, 'a'), (1, 2, 'a'), (1, 2, 'a')]
sage: G.degree(1)
7
sage: G.allow_loops(False)
```

```
sage: G.edges()
[(1, 2, 'a'), (1, 2, 'a'), (1, 2, 'a')]
sage: G.degree(1)
3
```

Ensure that trac ticket #13664 is fixed

```
sage: W = WeylGroup(["A",1])
sage: G = W.cayley_graph()
sage: Graph(G).degree()
[1, 1]
sage: h = Graph()
sage: h.add_edge(1,2,"a")
sage: h.add_edge(1,2,"a")
sage: h.degree()
[1, 1]
```

del_vertex(*v*)

Delete a vertex in `self`, failing silently if the vertex is not in the graph.

INPUT:

- v – vertex to be deleted.

OUTPUT:

- None.

See Also:

- `del_vertices()` – delete a bunch of vertices from this graph.

EXAMPLE:

```
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.

EXAMPLE:

```
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
sage: D.has_vertex(0)
True
```

depth_first_search(*v*, *reverse=False*, *ignore_direction=False*)

Returns 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
   (a) $v \leftarrow$ pop(S)
   (b) for each $w \in$ adj(v) do # for digraphs, use out-neighbor set oadj(v)
      i. if $D[w] = \infty$ then
         A. $D[w] \leftarrow D[v] + 1$
         B. push(S, w)
         C. 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(), implementation="c_graph")
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"]}, implementation="c_graph")
sage: list(G.depth_first_search("Frankfurt"))
['Frankfurt', 'Wurzburg', 'Nurnberg', 'Munchen', 'Kassel', 'Augsburg', 'Karlsruhe', 'Mannhei
```

has_vertex(*v*)

Returns whether v is a vertex of self.

INPUT:

- v – any object.

OUTPUT:

- True if v is a vertex of this graph; False otherwise.

EXAMPLE:

```
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
```

is_connected()

Returns whether the graph is connected.

EXAMPLES:

Petersen's graph is connected:

```
sage: DiGraph(graphs.PetersenGraph(),implementation="c_graph").is_connected()
True
```

While the disjoint union of two of them is not:

```
sage: DiGraph(2*graphs.PetersenGraph(),implementation="c_graph").is_connected()
False
```

A graph with non-integer vertex labels:

```
sage: Graph(graphs.CubeGraph(3), implementation='c_graph').is_connected()
True
```

is_directed_acyclic(*certificate=False*)

Returns whether the graph is both directed and acylic (possibly with a certificate)

INPUT:

- `certificate` – whether to return a certificate (`False` by default).

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],
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()

Returns 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]}, implementation="c_graph")
sage: g.is_strongly_connected()
True
```

But a transitive triangle is not:

```
sage: g = DiGraph({0: [1,2], 1: [2]}, implementation="c_graph")
sage: g.is_strongly_connected()
False
```

iterator_in_nbrs(v)

Returns 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.

EXAMPLE:

```
sage: P = DiGraph(graphs.PetersenGraph().to_directed(), implementation="c_graph")
sage: list(P._backend.iterator_in_nbrs(0))
[1, 4, 5]
```

iterator_nbrs(*v*)
Returns 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.

EXAMPLE:

```
sage: P = Graph(graphs.PetersenGraph(), implementation="c_graph")
sage: list(P._backend.iterator_nbrs(0))
[1, 4, 5]
```

iterator_out_nbrs(*v*)
Returns 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.

EXAMPLE:

```
sage: P = DiGraph(graphs.PetersenGraph().to_directed(), implementation="c_graph")
sage: list(P._backend.iterator_out_nbrs(0))
[1, 4, 5]
```

iterator_verts(*verts=None*)
Returns 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 an 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.

EXAMPLE:

```
sage: P = Graph(graphs.PetersenGraph(), implementation="c_graph")
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*)

Returns whether loops are allowed in this graph.

INPUT:

- `new` – (default: `None`); boolean (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`.

EXAMPLE:

```
sage: G = Graph(implementation='c_graph')
sage: G._backend.loops()
False
sage: G._backend.loops(True)
sage: G._backend.loops()
True
```

num_edges (*directed*)

Returns 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.

EXAMPLE:

```
sage: G = Graph(graphs.PetersenGraph(), implementation="c_graph")
sage: G._backend.num_edges(False)
15
```

TESTS:

Ensure that ticket #8395 is fixed.

```
sage: G = Graph({1:[1]}); G
Looped graph on 1 vertex
sage: G.edges(labels=False)
[(1, 1)]
```

```
sage: G.size()
1
sage: G = Graph({1:[2,2]}); G
Multi-graph on 2 vertices
sage: G.edges(labels=False)
[(1, 2), (1, 2)]
sage: G.size()
2
sage: G = Graph({1:[1,1]}); G
Looped multi-graph on 1 vertex
sage: G.edges(labels=False)
[(1, 1), (1, 1)]
sage: G.size()
2
sage: D = DiGraph({1:[1]}); D
Looped digraph on 1 vertex
sage: D.edges(labels=False)
[(1, 1)]
sage: D.size()
1
sage: D = DiGraph({1:[2,2], 2:[1,1]}); D
Multi-digraph on 2 vertices
sage: D.edges(labels=False)
[(1, 2), (1, 2), (2, 1), (2, 1)]
sage: D.size()
4
sage: D = DiGraph({1:[1,1]}); D
Looped multi-digraph on 1 vertex
sage: D.edges(labels=False)
[(1, 1), (1, 1)]
sage: D.size()
2
sage: from sage.graphs.base.sparse_graph import SparseGraphBackend
sage: S = SparseGraphBackend(7)
sage: S.num_edges(directed=False)
0
sage: S.loops(True)
sage: S.add_edge(1, 1, None, directed=False)
sage: S.num_edges(directed=False)
1
sage: S.multiple_edges(True)
sage: S.add_edge(1, 1, None, directed=False)
sage: S.num_edges(directed=False)
2
sage: from sage.graphs.base.dense_graph import DenseGraphBackend
sage: D = DenseGraphBackend(7)
sage: D.num_edges(directed=False)
0
sage: D.loops(True)
sage: D.add_edge(1, 1, None, directed=False)
sage: D.num_edges(directed=False)
1
```

num_verts()

Returns the number of vertices in `self`.

INPUT:

- None.

OUTPUT:

- The order of this graph.

See Also:

- `num_edges()` – return the number of (directed) edges in this graph.

EXAMPLE:

```
sage: G = Graph(graphs.PetersenGraph(), implementation="c_graph")
sage: G._backend.num_verts()
10
```

out_degree(*v*)

Returns the out-degree of 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`.

EXAMPLE:

```
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*)

Relabels 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(), implementation="c_graph")
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),
 (5, 9, None),
 (6, 7, None),
 (7, 8, None),
 (8, 9, None)]
```

shortest_path(*x*, *y*)

Returns 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.

OUTPUT:

- A list of vertices in the shortest path from x to y.

EXAMPLE:

```
sage: G = Graph(graphs.PetersenGraph(), implementation="c_graph")
sage: G.shortest_path(0, 1)
[0, 1]
```

shortest_path_all_vertices(*v*, *cutoff=None*)

Returns for each vertex u a shortest v−u path.

INPUT:

- v – a starting vertex in the shortest path.
- cutoff – maximal distance. Longer paths will not be returned.

OUTPUT:

- A list which associates to each vertex u the shortest path between u and v if there is one.

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([ len(paths[v]) == 0 or len(paths[v])-1 == g.distance(0,v) for v in g])
True
```

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.distar
True
```

strongly_connected_component_containing_vertex(*v*)

Returns 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.

EXAMPLE:

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

next()

x.next() -> the next value, or raise StopIteration

3.2 Fast sparse graphs

Fast sparse graphs

3.2.1 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._num_verts()
10
sage: S.add_vertex(9)
9
sage: S._num_verts()
10
```

But 10 is not, until we add it:

```
sage: S._num_verts()
10
sage: S.add_vertex(10)
10
sage: S._num_verts()
11
```

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.
sage: S._num_verts()
10
sage: S._num_arcs()
2
```

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)
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.

3.2.2 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(*u*, *v*)

Adds arc `(u, v)` to the graph with no label.

INPUT:

- `u, v` – non-negative integers, must be in self

EXAMPLE:

```
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_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

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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
```

NOTES:

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_all_arcs(*u*, *v*)

Deletes all arcs from `u` to `v`.

INPUT:

- `u, v` - integers

EXAMPLE:

```
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)
```

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

EXAMPLE:

```
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(*u*, *v*)
Checks whether arc `(u, v)` is in the graph.

INPUT:

- `u, v` - integers

EXAMPLE:

```
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_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

EXAMPLE:

```
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
```

in_neighbors(*v*)

Gives all u such that (u, v) is an arc of the graph.

INPUT:

- v - 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(3,1)
sage: G.add_arc(1,3)
sage: G.in_neighbors(1)
[0, 3]
sage: G.in_neighbors(3)
[1]
```

NOTE: Due to the implementation of SparseGraph, this method is much more expensive than neighbors_unsafe.

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
```

out_neighbors(*u*)

Gives all `v` such that `(u, v)` is an arc of the graph.

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_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

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**(*n*, *directed=True*)

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, implementation="c_graph", 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: G.vertices()
[0,
...
 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)`

EXAMPLE:

```
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)]
```

TESTS:

```
sage: D = DiGraph(implementation='c_graph', sparse=True)
sage: D.add_edge(0,1,2)
sage: D.add_edge(0,1,3)
sage: D.edges()
[(0, 1, 3)]
```

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)`

EXAMPLE:

```
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)`

EXAMPLE:

```
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)]
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)]
```

TESTS:

```
sage: G = Graph(implementation='c_graph', sparse=True)
sage: G.add_edge(0,1,2)
sage: G.delete_edge(0,1)
sage: G.edges()
[]

sage: G = Graph(multiedges=True, implementation='c_graph', sparse=True)
sage: G.add_edge(0,1,2)
sage: G.add_edge(0,1,None)
sage: G.delete_edge(0,1)
sage: G.edges()
[(0, 1, 2)]
```

Do we remove loops correctly? (trac ticket #12135):

```
sage: g=Graph({0:[0,0,0]}, implementation='c_graph', sparse=True)
sage: g.edges(labels=False)
[(0, 0), (0, 0), (0, 0)]
sage: g.delete_edge(0,0); g.edges(labels=False)
[(0, 0), (0, 0)]
```

get_edge_label(*u*, *v*)

Returns the edge label for `(u,v)`.

INPUT:

- `u,v` - the vertices of the edge

EXAMPLE:

```
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`

EXAMPLE:

```
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.

INPUT:

- `vertices` - a list of vertex labels
- `labels` - boolean, whether to return labels as well

EXAMPLE:

```
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)]
```

TEST:

```
sage: g = graphs.PetersenGraph()
sage: g.edges_incident([0,1,2])
[(0, 1, None),
 (0, 4, None),
```

```
 (0, 5, None),
 (1, 2, None),
 (1, 6, None),
 (2, 3, None),
 (2, 7, None)]
```

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

EXAMPLE:

```
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

EXAMPLE:

```
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)]
```

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`

EXAMPLE:

```
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')]
```

class sage.graphs.base.sparse_graph.**id_dict**

This is a helper class for pickling sparse graphs. It emulates a dictionary `d` which contains all objects, and always, `d[x] == x`.

EXAMPLE:

```
sage: from sage.graphs.base.sparse_graph import id_dict
sage: d = id_dict()
sage: d[None] is None
True
sage: d[7]
7
sage: d[{}]
{}
sage: d[()]
()
```

sage.graphs.base.sparse_graph.**random_stress**()

Randomly search for mistakes in the code.

DOCTEST (No output indicates that no errors were found):

```
sage: from sage.graphs.base.sparse_graph import random_stress
sage: for _ in xrange(20):
...       random_stress()
```

3.3 Fast dense graphs

Fast dense graphs

3.3.1 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._num_verts()
10
sage: D.add_vertex(9)
9
sage: D._num_verts()
10
```

But 10 is not, until we add it:

```
sage: D._num_verts()
10
sage: D.add_vertex(10)
10
sage: D._num_verts()
11
```

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
sage: D._num_verts()
10
sage: D._num_arcs()
2
```

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.

3.3.2 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 radix_div_shift
cdef int radix_mod_mask
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. The ints `radix_div_shift` and `radix_mod_mask` are simply for doing efficient division by powers of two, and `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: 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(*u*, *v*)

Adds arc (u, v) to the graph.

INPUT:

- u, v – non-negative integers, must be in self

EXAMPLE:

```
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
```

del_all_arcs(*u*, *v*)

Deletes the arc from u to v.

INPUT:

- u, v - integers

NOTE: The naming of this function is for consistency with SparseGraph. Of course, there can be at most one arc for a DenseGraph.

EXAMPLE:

```
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
```

has_arc(*u*, *v*)

Checks whether arc (u, v) is in the graph.

INPUT: u, v – integers

EXAMPLE:

```
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
```

in_neighbors(*v*)
Gives all u such that (u, v) is an arc of the graph.

INPUT:

- v - integer

EXAMPLES:

```
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]
```

out_neighbors(*u*)
Gives all v such that (u, v) is an arc of the graph.

INPUT:

- u - integer

EXAMPLES:

```
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]
sage: G.out_neighbors(1)
[2, 3]
```

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**(*n*, *directed=True*)

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, implementation="c_graph", 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: G.vertices()
[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*)

Adds the 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.

EXAMPLE:

```
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` - 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)`

EXAMPLE:

```
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.

EXAMPLE:

```
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*)
Returns the edge label for `(u,v)`. Always None, since dense graphs do not support edge labels.

INPUT:

- `u,v` - the vertices of the edge

EXAMPLE:

```
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),
 (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*)

Returns 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)

EXAMPLE:

```
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*)

Iterate 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 labels as well

EXAMPLE:

```
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*)

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

EXAMPLE:

```
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*)

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

EXAMPLE:

```
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`

EXAMPLE:

```
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.
```

sage.graphs.base.dense_graph.**random_stress**()

Randomly search for mistakes in the code.

DOCTEST (No output indicates that no errors were found):

```
sage: from sage.graphs.base.dense_graph import random_stress
sage: for _ in xrange(400):
...       random_stress()
```

3.4 Static dense graphs

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.

3.4.1 Index

Cython functions

`dense_graph_init`	Fills a binary matrix with the information of a (di)graph.

Python functions

`is_strongly_regular()`	Tests if a graph is strongly regular

3.4.2 Functions

sage.graphs.base.static_dense_graph.**is_strongly_regular**(*g*, *parameters=False*)

Tests whether `self` 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 Wikipedia article Strongly regular graph

INPUT:

- `parameters` (boolean) – 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)]).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)]).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().
```

3.5 Static Sparse Graphs

Static Sparse Graphs

3.5.1 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.

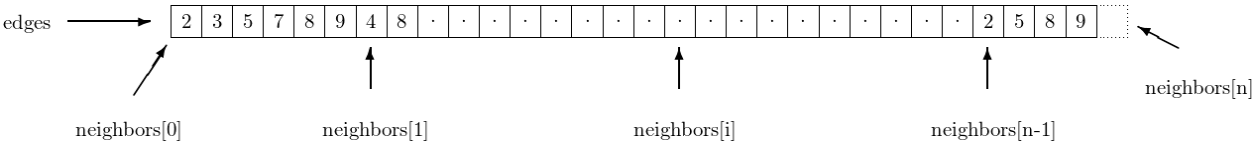
The data structure, however, is made to save the maximum amount of computations for graph algorithms whose main operation is to *list the out-neighbours of a vertex* (which is precisely what BFS, DFS, distance computations and the flow-related stuff waste their life on).

The code contained in this module is written C-style. While Sage needs a class for static graphs (not available today, i.e. 2012-01-13) it is not what we try to address here. The purpose is efficiency and simplicity.

Author:

- Nathann Cohen (2011)

3.5.2 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` – 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",str(i),"to",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 ! :-)

3.5.3 Technical details

- When creating a `fast_digraph` from a `Graph` or `DiGraph` named `G`, the i^{th} vertex corresponds to `G.vertices()[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.

3.5.4 Cython functions

`init_short_digraph(short_digraph g, G)`	Initializes `short_digraph g` from a Sage (Di)Graph.
`int n_edges(short_digraph g)`	Returns the number of edges in `g`
`int out_degree(short_digraph g, int i)`	Returns the out-degree of vertex i in `g`
`has_edge(short_digraph g, int u, int v)`	Tests the existence of an edge.
`edge_label(short_digraph g, int * edge)`	Returns the label associated with a given edge
`init_empty_copy(short_digraph dst, short_digraph src)`	Allocates `dst` so that it can contain as many vertices and edges as `src`.
`init_reverse(short_digraph dst, short_digraph src)`	Initializes `dst` to a copy of `src` with all edges in the opposite direction.
`free_short_digraph(short_digraph g)`	Free the ressources 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`.

3.5.5 What is this module used for ?

At the moment, it is only used in the `sage.graphs.distances_all_pairs` module.

3.5.6 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.`**`strongly_connected_components`**`(`G`)`

Returns the strongly connected components of the given DiGraph.

INPUT:

- `G` – a DiGraph.

Note: This method has been written as an attempt to solve the slowness reported in trac ticket #12235. It is not the one used by `sage.graphs.digraph.DiGraph.strongly_connected_components()` as saving some time on the computation of the strongly connected components is not worth copying the whole graph, but it is a nice way to test this module's functions. It is also tested in the doctest or `sage.graphs.digraph.DiGraph.strongly_connected_components()`.

EXAMPLE:

```
sage: from sage.graphs.base.static_sparse_graph import strongly_connected_components
sage: g = digraphs.ButterflyGraph(2)
sage: strongly_connected_components(g)
[[('00', 0)], [('00', 1)], [('00', 2)], [('01', 0)], [('01', 1)], [('01', 2)],
[('10', 0)], [('10', 1)], [('10', 2)], [('11', 0)], [('11', 1)], [('11', 2)]]
```

3.6 Static sparse graph backend

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`)

3.6.1 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.

3.6.2 Classes and methods

class `sage.graphs.base.static_sparse_backend.`**`StaticSparseBackend`**(*G*, *loops=False*, *multi-edges=False*)

Bases: `sage.graphs.base.c_graph.CGraphBackend`

A graph `backend` for static sparse graphs.

EXAMPLE:

```
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: gi.edges_incident('111')
[('111', '110', '0'), ('111', '111', '1'), ('111', '112', '2'), ('111', '113', '3')]
sage: sorted(g.edges()) == sorted(gi.edges())
True

sage: g = graphs.PetersenGraph()
sage: gi=Graph(g,data_structure="static_sparse")
sage: g == gi
True
sage: sorted(g.edges()) == sorted(gi.edges())
True

sage: gi = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, data_structure="sta
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}}, dat
sage: G == GI
True

sage: G = graphs.OddGraph(4)
sage: d = G.diameter()
sage: H = G.distance_graph(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()
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
```

allows_loops(*value=None*)

Returns 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`.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.allows_loops()
False
```

```
sage: g = StaticSparseBackend(graphs.PetersenGraph(), loops=True)
sage: g.allows_loops()
True
```

degree(*v*, *directed*)

Returns 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`.

EXAMPLE:

```
sage: g = Graph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.degree(0)
3
```

get_edge_label(*u*, *v*)

Returns the edge label for `(u,v)`.

INPUT:

- `u, v` – two vertices

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: print g.get_edge_label(0,1)
None
sage: print g.get_edge_label(0,"Hey")
Traceback (most recent call last):
...
LookupError: One of the two vertices does not belong to the graph
sage: print g.get_edge_label(0,7)
Traceback (most recent call last):
...
LookupError: The edge does not exist

sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(digraphs.DeBruijn(3,2))
sage: g.has_edge('00','01','1')
True
sage: g.has_edge('00','01','0')
False
```

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` – two vertices
- `l` – a label

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.has_edge(0,1,'e')
False
sage: g.has_edge(0,4,None)
True
```

has_vertex(*v*)

Tests if the vertex belongs to the graph

INPUT:

- v – a vertex (or not?)

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.has_vertex(0)
True
sage: g.has_vertex("Hey")
False
```

in_degree(*v*)

Returns the in-degree of a vertex

INPUT:

- v – a vertex

EXAMPLE:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.in_degree(0)
3
```

iterator_edges(*vertices*, *labels*)

Returns an iterator over the graph's edges.

INPUT:

- `vertices` – only returns the edges incident to at least one vertex of `vertices`.
- `labels` – whether to return edge labels too

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: list(g.iterator_edges(g.iterator_verts(None), False))
[(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)]
```

trac ticket #15665:

```
sage: Graph(immutable=True).edges()
[]
```

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

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: list(g.iterator_in_edges([0],False))
[(0, 1), (0, 4), (0, 5)]
sage: list(g.iterator_in_edges([0],True))
[(0, 1, None), (0, 4, None), (0, 5, None)]

sage: DiGraph(digraphs.Path(5),immutable=False).incoming_edges([2])
[(1, 2, None)]
sage: DiGraph(digraphs.Path(5),immutable=True).incoming_edges([2])
[(1, 2, None)]
```

iterator_in_nbrs(*v*)

Returns the out-neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLE:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.neighbors_in(0)
[1, 4, 5]
```

iterator_nbrs(*v*)

Returns the neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLE:

```
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

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: list(g.iterator_out_edges([0], False))
[(0, 1), (0, 4), (0, 5)]
sage: list(g.iterator_out_edges([0],True))
[(0, 1, None), (0, 4, None), (0, 5, None)]
```

iterator_out_nbrs(*v*)

Returns the out-neighbors of a vertex

INPUT:

- v – a vertex

EXAMPLE:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.neighbors_out(0)
[1, 4, 5]
```

iterator_verts(*vertices*)

Returns 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.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: list(g.iterator_verts(None))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list(g.iterator_verts([1,"Hey","I am a french fry"]))
[1]
```

multiple_edges(*value=None*)

Returns 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.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.multiple_edges()
False
sage: g = StaticSparseBackend(graphs.PetersenGraph(), multiedges=True)
sage: g.multiple_edges()
True
```

num_edges(*directed*)

Returns the number of edges

INPUT:

- directed (boolean) – whether to consider the graph as directed or not.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.num_edges(False)
15
```

Testing the exception:

```
sage: g = StaticSparseBackend(digraphs.Circuit(4))
sage: g.num_edges(False)
```

```
Traceback (most recent call last):
...
NotImplementedError: Sorry, I have no idea what is expected in this situation. I don't think
```

trac ticket #15491:

```
sage: g=digraphs.RandomDirectedGNP(10,.3)
sage: gi=DiGraph(g,data_structure="static_sparse")
sage: gi.size() == len(gi.edges())
True
```

num_verts()

Returns the number of vertices

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.num_verts()
10
```

out_degree(*v*)

Returns the out-degree of a vertex

INPUT:

- v – a vertex

EXAMPLE:

```
sage: g = DiGraph(graphs.PetersenGraph(), data_structure="static_sparse")
sage: g.out_degree(0)
3
```

relabel(*perm*, *directed*)

Relabel the graphs' vertices. No way.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseBackend
sage: g = StaticSparseBackend(graphs.PetersenGraph())
sage: g.relabel([],True)
Traceback (most recent call last):
...
ValueError: Thou shalt not relabel an immutable graph
```

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*)

Adds a vertex to the graph. No way.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.add_vertex(45)
Traceback (most recent call last):
...
ValueError: Thou shalt not add a vertex to an immutable graph
```

del_vertex(*k*)
Removes a vertex from the graph. No way.

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.del_vertex(45)
Traceback (most recent call last):
...
ValueError: Thou shalt not remove a vertex from an immutable graph
```

has_arc(*u*, *v*)
Tests if uv is an edge of the graph

INPUT:

- u, v – integers

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.has_arc(0,1)
True
sage: g.has_arc(0,7)
False
```

has_vertex(*n*)
Tests if a vertex belongs to the graph

INPUT:

- n – an integer

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.has_vertex(1)
True
sage: g.has_vertex(10)
False
```

in_degree(*u*)
Returns the in-degree of a vertex

INPUT:

- u – a vertex

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.in_degree(0)
3
```

in_neighbors(*u*)
Returns the in-neighbors of a vertex

INPUT:

- u – a vertex

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.in_neighbors(0)
[1, 4, 5]
```

out_degree(*u*)

Returns the out-degree of a vertex

INPUT:

- u – a vertex

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.out_degree(0)
3
```

out_neighbors(*u*)

List the out-neighbors of a vertex

INPUT:

- u – a vertex

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.out_neighbors(0)
[1, 4, 5]
```

verts()

Returns the list of vertices

TEST:

```
sage: from sage.graphs.base.static_sparse_backend import StaticSparseCGraph
sage: g = StaticSparseCGraph(graphs.PetersenGraph())
sage: g.verts()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.7 Implements various backends for Sage graphs.

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.

TESTS:

```
sage: import sage.graphs.base.graph_backends
```

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
```

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.add_edge(1,2,'a',True)
Traceback (most recent call last):
...
NotImplementedError
```

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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.add_edges([],True)
Traceback (most recent call last):
...
NotImplementedError
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.add_vertex(0)
Traceback (most recent call last):
...
NotImplementedError
```

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*)

Returns the total number of vertices incident to v.

INPUT:

- v – a vertex label
- directed – boolean

OUTPUT:

degree of v

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.degree(1, False)
Traceback (most recent call last):
...
NotImplementedError
```

del_edge(*u*, *v*, *l*, *directed*)

Deletes the edge (u,v) with label l.

INPUT:

- u, v – vertices
- l – edge label
- directed – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.del_edge(1,2,'a',True)
Traceback (most recent call last):
...
NotImplementedError
```

del_vertex(*v*)

Delete a labelled vertex in self.

INPUT:

- v – vertex label

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.del_vertex(0)
Traceback (most recent call last):
...
NotImplementedError
```

del_vertices(*vertices*)

Delete labelled vertices in self.

INPUT:

- vertices – iterator of vertex labels

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.del_vertices([1,2,3])
Traceback (most recent call last):
...
NotImplementedError
```

get_edge_label(*u*, *v*)

Returns the edge label of (u,v).

INPUT:

- u, v – vertex labels

OUTPUT:

label of (u,v)

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.get_edge_label(1,2)
Traceback (most recent call last):
...
NotImplementedError
```

has_edge(*u*, *v*, *l*)

True if self has an edge (u,v) with label l.

INPUT:

- u, v – vertex labels
- l – label

OUTPUT:

boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.has_edge(1,2,'a')
Traceback (most recent call last):
...
NotImplementedError
```

has_vertex(*v*)

True if self has a vertex with label v.

INPUT:

- v – vertex label

OUTPUT: boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.has_vertex(0)
Traceback (most recent call last):
...
NotImplementedError
```

iterator_edges(*vertices*, *labels*)

Iterate over the edges incident to a sequence of vertices. Edges are assumed to be undirected.

INPUT:

- `vertices` – a list of vertex labels
- `labels` – boolean

OUTPUT:

a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_edges([],True)
Traceback (most recent call last):
...
NotImplementedError
```

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

OUTPUT: a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_in_edges([],True)
Traceback (most recent call last):
...
NotImplementedError
```

iterator_in_nbrs(*v*)

Iterate 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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_in_nbrs(0)
Traceback (most recent call last):
...
NotImplementedError
```

iterator_nbrs(*v*)
Iterate over the vertices adjacent to v.

INPUT:

- v – vertex label

OUTPUT:

a generator which yields vertex labels

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_nbrs(0)
Traceback (most recent call last):
...
NotImplementedError
```

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

OUTPUT:

a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_out_edges([],True)
Traceback (most recent call last):
...
NotImplementedError
```

iterator_out_nbrs(*v*)
Iterate 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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_out_nbrs(0)
Traceback (most recent call last):
...
NotImplementedError
```

iterator_verts(*verts*)
Iterate over the vertices v with labels in verts.

INPUT:

- vertex – vertex labels

OUTPUT:

a generator which yields vertices

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.iterator_verts(0)
Traceback (most recent call last):
...
NotImplementedError
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.loops(True)
Traceback (most recent call last):
...
NotImplementedError
sage: G.loops(None)
Traceback (most recent call last):
...
NotImplementedError
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.multiple_edges(True)
Traceback (most recent call last):
...
NotImplementedError
sage: G.multiple_edges(None)
Traceback (most recent call last):
...
NotImplementedError
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.name("A Generic Graph")
```

```
Traceback (most recent call last):
...
NotImplementedError
sage: G.name(None)
Traceback (most recent call last):
...
NotImplementedError
```

num_edges(*directed*)

The number of edges in self

INPUT:

- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.num_edges(True)
Traceback (most recent call last):
...
NotImplementedError
sage: G.num_edges(False)
Traceback (most recent call last):
...
NotImplementedError
```

num_verts()

The number of vertices in self

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.num_verts()
Traceback (most recent call last):
...
NotImplementedError
```

relabel(*perm*, *directed*)

Relabel the vertices of self by a permutation.

INPUT:

- `perm` – permutation
- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.relabel([],False)
Traceback (most recent call last):
...
NotImplementedError
```

set_edge_label(*u*, *v*, *l*, *directed*)

Label the edge (u,v) by l.

INPUT:

- `u, v` – vertices
- `l` – edge label

- directed – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.GenericGraphBackend()
sage: G.set_edge_label(1,2,'a',True)
Traceback (most recent call last):
...
NotImplementedError
```

class sage.graphs.base.graph_backends.**NetworkXDiGraphDeprecated**

Bases: sage.structure.sage_object.SageObject

Class for unpickling old networkx.XDiGraph formats

TESTS:

```
sage: import sage.graphs.base.graph_backends
```

mutate()

Change the old networkx XDiGraph format into the new one.

OUTPUT:

- The networkx.DiGraph or networkx.MultiDiGraph corresponding to the unpickled data.

EXAMPLES:

```
sage: from sage.graphs.base.graph_backends import NetworkXDiGraphDeprecated as NXDGD
sage: X = NXDGD()
doctest:...
sage: X.adj = {1:{2:7}, 2:{1:[7,8], 3:[4,5,6,7]}}
sage: X.multiedges = True
sage: G = X.mutate()
sage: G.edges()
[(1, 2), (2, 1), (2, 3)]
sage: G.edges(data=True)
[(1, 2, {'weight': 7}), (2, 1, {8: {}, 7: {}}), (2, 3, {4: {}, 5: {}, 6: {}, 7: {}})]
```

class sage.graphs.base.graph_backends.**NetworkXGraphBackend**(*N=None*)

Bases: sage.graphs.base.graph_backends.GenericGraphBackend

A wrapper for NetworkX as the backend of a graph.

TESTS:

```
sage: import sage.graphs.base.graph_backends
```

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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.add_edge(1,2,'a',True)
```

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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.add_edges([],True)
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.add_vertex(0)
```

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

degree(*v*, *directed*)
Returns the total number of vertices incident to v.

INPUT:

- `v` – a vertex label
- `directed` – boolean

OUTPUT:

degree of v

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.add_vertices(range(3))
sage: G.degree(1, False)
0
```

del_edge(*u*, *v*, *l*, *directed*)
Deletes the edge (u,v) with label l.

INPUT:

- `u, v` – vertices
- `l` – edge label
- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.del_edge(1,2,'a',True)
```

del_vertex(*v*)
Delete a labelled vertex in self.

INPUT:

- `v` – vertex label

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.del_vertex(0)
Traceback (most recent call last):
...
NetworkXError: The node 0 is not in the graph.
```

del_vertices(*vertices*)
Delete labelled vertices in self.

INPUT:

- `vertices` – iterator of vertex labels

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.del_vertices([1,2,3])
Traceback (most recent call last):
...
NetworkXError: The node 1 is not in the graph.
```

get_edge_label(*u*, *v*)
Returns the edge label of (u,v).

INPUT:

- `u, v` – vertex labels

OUTPUT: label of (u,v)

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.get_edge_label(1,2)
Traceback (most recent call last):
...
NetworkXError: Edge (1,2) requested via get_edge_label does not exist.
```

has_edge(*u*, *v*, *l*)

True if self has an edge (u,v) with label l.

INPUT:

- u, v – vertex labels
- l – label

OUTPUT: boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.has_edge(1,2,'a')
False
```

has_vertex(*v*)

True if self has a vertex with label v.

INPUT:

- v – vertex label

OUTPUT: boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.has_vertex(0)
False
```

iterator_edges(*vertices*, *labels*)

Iterate over the edges incident to a sequence of vertices. Edges are assumed to be undirected.

INPUT:

- vertices – a list of vertex labels
- labels – boolean

OUTPUT: a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.iterator_edges([],True)
<generator object iterator_edges at ...>
```

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

OUTPUT: a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: i = G.iterator_in_edges([],True)
```

iterator_in_nbrs(*v*)

Iterate 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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.iterator_in_nbrs(0)
Traceback (most recent call last):
...
AttributeError: 'MultiGraph' object has no attribute 'predecessors_iter'
```

iterator_nbrs(*v*)

Iterate over the vertices adjacent to v.

INPUT:

- v – vertex label

OUTPUT: a generator which yields vertex labels

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.add_vertex(0)
sage: G.iterator_nbrs(0)
<dictionary-keyiterator object at ...>
```

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

OUTPUT: a generator which yields edges, with or without labels depending on the labels parameter.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: i = G.iterator_out_edges([],True)
```

iterator_out_nbrs(*v*)

Iterate 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

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.iterator_out_nbrs(0)
Traceback (most recent call last):
...
AttributeError: 'MultiGraph' object has no attribute 'successors_iter'
```

iterator_verts(*verts*)

Iterate over the vertices v with labels in verts.

INPUT:

- vertex – vertex labels

OUTPUT: a generator which yields vertices

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.iterator_verts(0)
<generator object bunch_iter at ...>
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.loops(True)
sage: G.loops(None)
True
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.multiple_edges(True)
sage: G.multiple_edges(None)
True
```

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.

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.name("A NetworkX Graph")
sage: G.name(None)
'A NetworkX Graph'
```

num_edges (*directed*)

The number of edges in self

INPUT:

- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.num_edges(True)
0
sage: G.num_edges(False)
0
```

num_verts ()

The number of vertices in self

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.num_verts()
0
```

relabel (*perm*, *directed*)

Relabel the vertices of self by a permutation.

INPUT:

- `perm` – permutation
- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.relabel([],False)
```

set_edge_label (*u*, *v*, *l*, *directed*)

Label the edge (u,v) by l.

INPUT:

- `u,v` – vertices
- `l` – edge label
- `directed` – boolean

TESTS:

```
sage: G = sage.graphs.base.graph_backends.NetworkXGraphBackend()
sage: G.set_edge_label(1,2,'a',True)
```

class sage.graphs.base.graph_backends.**NetworkXGraphDeprecated**

Bases: sage.structure.sage_object.SageObject

Class for unpickling old networkx.XGraph formats

TESTS:

```
sage: from sage.graphs.base.graph_backends import NetworkXGraphDeprecated as NXGD
sage: X = NXGD()
doctest:...
```

mutate()

Change the old networkx XGraph format into the new one.

OUTPUT:

- The networkx.Graph or networkx.MultiGraph corresponding to the unpickled data.

EXAMPLES:

```
sage: from sage.graphs.base.graph_backends import NetworkXGraphDeprecated as NXGD
sage: X = NXGD()
doctest:...
sage: X.adj = {1:{2:7}, 2:{1:7}, 3:{2:[4,5,6,7]}, 2:{3:[4,5,6,7]}}
sage: X.multiedges = True
sage: G = X.mutate()
sage: G.edges()
[(1, 2), (2, 3)]
sage: G.edges(data=True)
[(1, 2, {'weight': 7}), (2, 3, {4: {}, 5: {}, 6: {}, 7: {}})]
```

CHAPTER

FOUR

HYPERGRAPHS

4.1 Hypergraph generators

At the moment this module only implement one method, which calls Brendan McKay's Nauty (http://cs.anu.edu.au/~bdm/nauty/) to enumerate hypergraphs up to isomorphism.

class sage.graphs.hypergraph_generators.**HypergraphGenerators**

A class consisting of constructors for common hypergraphs.

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*, *options=''*, *debug=False*)

Enumerates hypergraphs up to isomorphism using Nauty.

INPUT:

- `number_of_sets`, `number_of_vertices` (integers)
- `multiple_sets` (boolean) – whether to allow several sets of the hypergraph to be equal (set to `False` by default).
- `vertex_min_degree`, `vertex_max_degree` (integers) – define the maximum and minimum degree of an element from the ground set (i.e. the number of sets which contain it). Set to `None` by default.
- `set_min_size`, `set_max_size` (integers) – define the maximum and minimum size of a set. Set to `None` by default.
- `regular` (integer) – if set to an integer value k, requires the hypergraphs to be k-regular. It is actually a shortcut for the corresponing min/max values.
- `uniform` (integer) – if set to an integer value k, requires the hypergraphs to be k-uniform. It is actually a shortcut for the corresponing min/max values.
- `max_intersection` (integer) – constraints the maximum cardinality of the intersection of two sets fro the hypergraphs. Set to `None` by default.
- `connected` (boolean) – whether to require the hypergraphs to be connected. Set to `False` by default.
- `debug` (boolean) – 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/.

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)) # optional - nauty
[((), (0,), (1,), (0, 1))]
```

Only connected ones:

```
sage: list(hypergraphs.nauty(2,2, connected = True)) # optional - nauty
[((0,), (0, 1))]
```

Non-empty sets only:

```
sage: list(hypergraphs.nauty(3,2, set_min_size = 1)) # optional - nauty
[((0,), (1,), (0, 1))]
```

The Fano Plane, as the only 3-uniform hypergraph with 7 sets and 7 vertices:

```
sage: fano = hypergraphs.nauty(7,7, uniform = 3, max_intersection =1).next() # optional - na
sage: print fano # optional - nauty
((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 = hypergraphs.nauty(7,7, regular = 3, max_intersection =1).next() # optional - na
sage: print fano # optional - nauty
((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]

5. Assmus, J. Key, Designs and their codes, CUP, 1992.

ground_set()	Return the ground set (i.e the list of points).
num_points()	Return the size of the ground set.
num_blocks()	Return the number of blocks.
blocks()	Return the list of blocks.
block_sizes()	Return the set of block sizes.
degree()	Return the degree of a point p
is_connected()	Test whether the design is connected.
is_simple()	Test whether this design is simple (i.e. no repeated block).
incidence_matrix()	Return the incidence matrix A of the design
incidence_graph()	Return the incidence graph of the design
packing()	Return a maximum packing
is_t_design()	Test whether self is a $t-(v,k,l)$ design.
dual()	Return the dual design.
automorphism_group()	Return the automorphism group
edge_coloring()	Return an optimal edge coloring‘

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*, *test=None*, *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: 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: designs.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: designs.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 = designs.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 = designs.IncidenceStructure([0,1,2], [[0],[0,2]])
sage: B = designs.IncidenceStructure([1,0,2], [[0],[2,0]])
sage: B == A
True

sage: C = designs.BlockDesign(2, [[0], [1,0]])
sage: D = designs.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 = designs.IncidenceStructure(3, blocks, copy=False)
sage: I.blocks(copy=False) 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
Incidence structure with 7 points and 7 blocks
sage: G = P.automorphism_group()
sage: G.is_isomorphic(PGL(3,2))
True
sage: G
Permutation Group with generators [(2,3)(4,5), (2,4)(3,5), (1,2)(4,6), (0,1)(4,5)]
```

A non self-dual example:

```
sage: IS = designs.IncidenceStructure(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 = designs.IncidenceStructure('abc', ('ab','ac','bc'))
sage: I.automorphism_group()
Permutation Group with generators [('b','c'), ('a','b')]
sage: designs.IncidenceStructure([[(1,2),(3,4)]]).automorphism_group()
Permutation Group with generators [((1,2),(3,4))]
```

block_design_checker(*t*, *v*, *k*, *lmbda*, *type=None*)

This method is deprecated and will soon be removed (see trac ticket #16553). You could use `is_t_design()` instead.

This is *not* a wrapper for GAP Design's IsBlockDesign. The GAP Design function IsBlockDesign http://www.gap-system.org/Manuals/pkg/design/htm/CHAP004.htm apparently simply checks the record structure and no mathematical properties. Instead, the function below checks some necessary (but not sufficient) "easy" identities arising from the identity.

INPUT:

- `t` - the t as in "t-design"
- `v` - the number of points
- `k` - the number of blocks incident to a point
- `lmbda` - each t-tuple of points should be incident with lmbda blocks
- `type` - can be 'simple' or 'binary' or 'connected' Depending on the option, this wraps IsBinaryBlockDesign, IsSimpleBlockDesign, or IsConnectedBlockDesign.
  - –Binary: no block has a repeated element.
  - –Simple: no block is repeated.
  - –Connected: its incidence graph is a connected graph.

WARNING: This is very fast but can return false positives.

EXAMPLES:

```
sage: BD = designs.IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,
sage: BD.is_t_design(return_parameters=True)
(True, (2, 7, 3, 1))
```

```
sage: BD.block_design_checker(2, 7, 3, 1)
doctest:...: DeprecationWarning: .block_design_checker(v,t,k,lmbda) is deprecated; please us
.is_t_design(v,t,k,lmbda) instead
See http://trac.sagemath.org/16553 for details.
True

sage: BD.block_design_checker(2, 7, 3, 1,"binary")
doctest:...: DeprecationWarning: .block_design_checker(type='binary') is
deprecated; use .is_binary() instead
See http://trac.sagemath.org/16553 for details.
True

sage: BD.block_design_checker(2, 7, 3, 1,"connected")
doctest:...: DeprecationWarning: block_design_checker(type='connected') is
deprecated, please use .is_connected() instead
See http://trac.sagemath.org/16553 for details.
True

sage: BD.block_design_checker(2, 7, 3, 1,"simple")
doctest:...: DeprecationWarning: .block_design_checker(type='simple')
is deprecated; all designs here are simple!
See http://trac.sagemath.org/16553 for details.
True
```

block_sizes()

Return the set of block sizes.

EXAMPLES:

```
sage: BD = designs.IncidenceStructure(8, [[0,1,3],[1,4,5,6],[1,2],[5,6,7]])
sage: BD.block_sizes()
[3, 2, 4, 3]
sage: BD = designs.IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

blocks(*copy=True*)

Return the list of blocks.

INPUT:

- `copy` (boolean) – `True` by default. When set to `False`, a pointer toward the object's internal data is given. Set it to `False` only if you know what you are doing.

EXAMPLES:

```
sage: BD = designs.IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,
sage: BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

What you should pay attention to:

```
sage: blocks = BD.blocks(copy=False)
sage: del blocks[0:6]
sage: BD
Incidence structure with 7 points and 1 blocks
```

degree(*p=None*)

Return the degree of a point `p`

The degree of a point p is the number of blocks that contain it.

INPUT:

- p – a point. If set to None (default), a dictionary associating the points with their degrees is returned.

EXAMPLES:

```
sage: designs.steiner_triple_system(9).degree(3)
4
sage: designs.steiner_triple_system(9).degree()
{0: 4, 1: 4, 2: 4, 3: 4, 4: 4, 5: 4, 6: 4, 7: 4, 8: 4}
```

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))
```

TESTS:

```
sage: D = designs.IncidenceStructure(4, [[0,2],[1,2,3],[2,3]])
sage: D
Incidence structure with 4 points and 3 blocks
sage: D.dual()
Incidence structure with 3 points and 4 blocks
sage: print D.dual(algorithm="gap")        # optional - gap_packages
IncidenceStructure<points=[0, 1, 2], blocks=[[0], [0, 1, 2], [1], [1, 2]]>
sage: blocks = [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
sage: BD = designs.IncidenceStructure(7, blocks, name="FanoPlane");
sage: BD
Incidence structure with 7 points and 7 blocks
sage: print BD.dual(algorithm="gap")          # optional - gap_packages
IncidenceStructure<points=[0, 1, 2, 3, 4, 5, 6], blocks=[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1
sage: BD.dual()
Incidence structure with 7 points and 7 blocks
```

REFERENCE:

- Soicher, Leonard, Design package manual, available at http://www.gap-system.org/Manuals/pkg/design/htm/CHAP003.htm

dual_design(**args*, ***kwds*)

Deprecated: Use dual() instead. See trac ticket #16553 for details.

dual_incidence_structure(**args*, ***kwds*)

Deprecated: Use dual() instead. See trac ticket #16553 for details.

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(*copy=True*)

Return the ground set (i.e the list of points).

INPUT:

- `copy` (boolean) – `True` by default. When set to `False`, a pointer toward the object's internal data is given. Set it to `False` only if you know what you are doing.

EXAMPLES:

```
sage: designs.IncidenceStructure(3, [[0,1],[0,2]]).ground_set()
[0, 1, 2]
```

incidence_graph()

Return the incidence graph of the design, where the incidence matrix of the design is the adjacency matrix of the graph.

EXAMPLE:

```
sage: BD = designs.IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2
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
```

REFERENCE:

- Sage Reference Manual on Graphs

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 = designs.IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2
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 = designs.IncidenceStructure('abc', ('ab','abc','ac','c'))
sage: I.incidence_matrix()
[1 1 1 0]
[1 1 0 0]
[0 1 1 1]
```

is_block_design(**args*, ***kwds*)
Deprecated: Use `is_t_design()` instead. See trac ticket #16553 for details.

is_connected()
Test whether the design is connected.

EXAMPLES:

```
sage: designs.IncidenceStructure(3, [[0,1],[0,2]]).is_connected()
True
sage: designs.IncidenceStructure(4, [[0,1],[2,3]]).is_connected()
False
```

is_simple()
Test whether this design is simple (i.e. no repeated block).

EXAMPLES:

```
sage: designs.IncidenceStructure(3, [[0,1],[1,2],[0,2]]).is_simple()
True
sage: designs.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 = designs.IncidenceStructure(V, B)
sage: I.is_simple()
True
sage: I2 = designs.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 = designs.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 = designs.IncidenceStructure(8, blocks)
sage: S4_8.is_t_design(3,8,4,1)
True

sage: blocks = designs.steiner_quadruple_system(14)
sage: S4_14 = designs.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 = designs.IncidenceStructure(4,[[],[]])
sage: D.is_t_design(return_parameters=True)
(True,  (0, 4, 0, 2))

sage: D = designs.IncidenceStructure(4, [[0,1],[0,2],[0,3]])
sage: D.is_t_design(return_parameters=True)
(True, (0, 4, 2, 3))

sage: D = designs.IncidenceStructure(4, [[0],[1],[2],[3]])
```

```
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 1, 1))

sage: D = designs.IncidenceStructure(4,[[0,1],[2,3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 2, 1))

sage: D = designs.IncidenceStructure(4, [range(4)])
sage: D.is_t_design(return_parameters=True)
(True, (4, 4, 4, 1))
```

TESTS:

```
sage: blocks = designs.steiner_quadruple_system(8)
sage: S4_8 = designs.IncidenceStructure(8, blocks)
sage: R = range(15)
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(3,v,k,l)]
[(8, 4, 1)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(2,v,k,l)]
[(8, 4, 3)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(1,v,k,l)]
[(8, 4, 7)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(0,v,k,l)]
[(8, 4, 14)]
sage: A = designs.AffineGeometryDesign(3, 1, GF(2))
sage: A.is_t_design(return_parameters=True)
(True, (2, 8, 2, 1))
sage: A = designs.AffineGeometryDesign(4, 2, GF(2))
sage: A.is_t_design(return_parameters=True)
(True, (3, 16, 4, 1))
sage: I = designs.IncidenceStructure(2, [])
sage: I.is_t_design(return_parameters=True)
(True, (0, 2, 0, 0))
sage: I = designs.IncidenceStructure(2, [[0],[0,1]])
sage: I.is_t_design(return_parameters=True)
(False, (0, 0, 0, 0))
```

num_blocks()

Return the number of blocks.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_blocks()
7
sage: B = designs.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 = designs.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. Only useful when `algorithm == "LP"`.

EXAMPLE:

```
sage; IncidenceStructure([[1,2],[3,"A"],[2,3]]).packing()
[[1, 2], [3, 'A']]
sage: len(designs.steiner_triple_system(9).packing())
3
```

parameters()

Deprecated function. You should use `is_t_design()` instead.

EXAMPLES:

```
sage: I = designs.IncidenceStructure('abc', ['ab','ac','bc'])
sage: I.parameters()
doctest:...: DeprecationWarning: .parameters() is deprecated. Use
`is_t_design` instead
See http://trac.sagemath.org/16553 for details.
(2, 3, 2, 1)
```

points(**args*, ***kwds*)

Deprecated: Use `ground_set()` instead. See trac ticket #16553 for details.

sage.combinat.designs.incidence_structures.**IncidenceStructureFromMatrix**(*M*, *name=None*)

Deprecated function that builds an incidence structure from a matrix.

You should now use `designs.IncidenceStructure(incidence_matrix=M)`.

INPUT:

- `M` – a binary matrix. Creates a set of "points" from the rows and a set of "blocks" from the columns.

EXAMPLES:

```
sage: BD1 = designs.IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5
sage: M = BD1.incidence_matrix()
sage: BD2 = IncidenceStructureFromMatrix(M)
doctest:...: DeprecationWarning: IncidenceStructureFromMatrix is deprecated.
Please use designs.IncidenceStructure(incidence_matrix=M) instead.
See http://trac.sagemath.org/16553 for details.
sage: BD1 == BD2
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()`	Computes all n-colorings a graph
`first_coloring()`	Returns the first vertex coloring found
`number_of_n_colorings()`	Computes the number of n-colorings of a graph
`numbers_of_colorings()`	Computes the number of colorings of a graph
`chromatic_number()`	Returns the chromatic number of the graph
`vertex_coloring()`	Computes Vertex colorings and chromatic numbers

Other colorings

`grundy_coloring()`	Computes Grundy numbers and Grundy colorings
`b_coloring()`	Computes a b-chromatic numbers and b-colorings
`edge_coloring()`	Compute chromatic index and edge colorings
`round_robin()`	Computes a round-robin coloring of the complete graph on n vertices
`linear_arboricity()`	Computes the linear arboricity of the given graph
`acyclic_edge_coloring()`	Computes 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`**

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*)

Calls `self.random_all_graph_colorings()`. In the future, if other methods are added, it should call them, too.

TESTS:

```
sage: from sage.graphs.graph_coloring import Test
sage: Test().random(1)
```

random_all_graph_colorings(*tests=1000*)

Verifies 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.

TESTS:

```
sage: from sage.graphs.graph_coloring import Test
sage: Test().random_all_graph_colorings(1)
```

sage.graphs.graph_coloring.**acyclic_edge_coloring**(*g*, *hex_colors=False*, *value_only=False*, *k=0*, *solver=None*, *verbose=0*)

Computes 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)

–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)

–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. Set to 0 by default, which means quiet.

ALGORITHM:

Linear Programming

EXAMPLE:

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.edges(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*)

Computes all n-colorings of the graph G by casting the graph coloring problem into an exact cover problem, and passing 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
- $count_only$ **– (default: `False`) when set to `True`, it returns 1** for each coloring
- hex_colors – (default: `False`) when set to `False`, it labels the colors [0,1,..,‘‘n‘‘-1], otherwise it uses the RGB Hex labeling
- $vertex_color_dict$ – (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.
```

TESTS:

```
sage: G = Graph({0:[1,2,3],1:[2]})
sage: for C in all_graph_colorings(G,0): print C
sage: for C in all_graph_colorings(G,-1): print C
Traceback (most recent call last):
...
ValueError: n must be non-negative.
sage: G = Graph({0:[1],1:[2]})
sage: for c in all_graph_colorings(G,2, vertex_color_dict = True): print c
{0: 0, 1: 1, 2: 0}
{0: 1, 1: 0, 2: 1}
sage: for c in all_graph_colorings(G,2,hex_colors = True): print c
{'#00ffff': [1], '#ff0000': [0, 2]}
{'#ff0000': [1], '#00ffff': [0, 2]}
sage: for c in all_graph_colorings(G,2,hex_colors=True,vertex_color_dict = True): print c
{0: '#ff0000', 1: '#00ffff', 2: '#ff0000'}
{0: '#00ffff', 1: '#ff0000', 2: '#00ffff'}
```

```
sage: for c in all_graph_colorings(G, 2, vertex_color_dict = True): print c
{0: 0, 1: 1, 2: 0}
{0: 1, 1: 0, 2: 1}
sage: for c in all_graph_colorings(G, 2, count_only=True, vertex_color_dict = True): print c
1
1
```

sage.graphs.graph_coloring.**b_coloring**(*g*, *k*, *value_only=True*, *solver=None*, *verbose=0*)

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 ideia. 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.

An 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 $m(G) = max\ \{k | `there are k vertices of degree at least` k-1\}$, 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
- `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`.
- `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.
- `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*)

Returns 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*)

Properly colors the edges of a graph. See the URL http://en.wikipedia.org/wiki/Edge_coloring for further details on edge coloring.

INPUT:

- `g` – a graph.
- `value_only` – (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` – (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` – (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 !

EXAMPLE:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: g = graphs.PetersenGraph()
sage: edge_coloring(g, value_only=True)
4
```

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
```

sage.graphs.graph_coloring.**first_coloring**(*G*, *n=0*, *hex_colors=False*)

Given a graph, and optionally a natural number n, returns the first coloring we find with at least n colors.

INPUT:

- `hex_colors` – (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).
- `n` – The minimal number of colors to try.

EXAMPLES:

```
sage: from sage.graphs.graph_coloring import first_coloring
sage: G = Graph({0: [1, 2, 3], 1: [2]})
sage: first_coloring(G, 3)
[[1, 3], [0], [2]]
```

sage.graphs.graph_coloring.**grundy_coloring**(*g*, *k*, *value_only=True*, *solver=None*, *verbose=0*)

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.

INPUT:

- `k` (integer) – Maximum 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`.
- `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.
- `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*)

Computes 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, $la(G) \geq \lceil \frac{\Delta(G)}{2} \rceil$.

It is conjectured in [Aki80] that $la(G) \leq \lceil \frac{\Delta(G)+1}{2} \rceil$.

INPUT:

- `hex_colors` (boolean)
  - 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)
  - 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`
- `plus_one` (integer) – whether to use $\lceil \frac{\Delta(G)}{2} \rceil$ or $\lceil \frac{\Delta(G)+1}{2} \rceil$ colors.
  - If `0`, computes a decomposition of G into $\lceil \frac{\Delta(G)}{2} \rceil$ forests of paths
  - If `1`, computes a decomposition of G into $\lceil \frac{\Delta(G)+1}{2} \rceil$ colors, which is the conjectured general bound.
  - If `plus_one = None` (default), 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. Set** to 0 by default, which means quiet.

ALGORITHM:

Linear Programming

COMPLEXITY:

NP-Hard

EXAMPLE:

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.GridGraph([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.edges(labels = None)])
True
```

REFERENCES:

`sage.graphs.graph_coloring.`**`number_of_n_colorings`**`(`G, n`)`

Computes the number of n-colorings of a graph

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`)`

Returns the number of n-colorings of the graph G for 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`)`

Computes 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*)

Computes the chromatic number of the given graph or tests its k-colorability. See http://en.wikipedia.org/wiki/Graph_coloring for further details on graph coloring.

INPUT:

- g – a graph.
- k – (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 – (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 – (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.

EXAMPLE:

```
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)

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 (#11252)
- Nico Van Cleemput (2013-05-27): Handle the empty graph (#14525)

REFERENCE:

5.2.1 Methods

`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 [NisOst2003].

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]]
```

TEST:

```
sage: g = Graph()
sage: g.cliques_maximum()
[[]]
```

sage.graphs.cliquer.**clique_number**(*graph*)

Returns the size of the largest clique of the graph (clique number).

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: clique_number(C)
4
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: clique_number(G)
3
```

TEST:

```
sage: g = Graph()
sage: g.clique_number()
0
```

sage.graphs.cliquer.**list_composition**(*a*, *b*)

Composes a list `a` with a map `b`.

EXAMPLES:

```
sage: from sage.graphs.cliquer import list_composition
sage: list_composition([1,3,'a'], {'a':'b', 1:2, 2:3, 3:4})
[2, 4, 'b']
```

sage.graphs.cliquer.**max_clique**(*graph*)

Returns the vertex set of a maximum complete subgraph.

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: C=graphs.PetersenGraph()
sage: max_clique(C)
[7, 9]
```

TEST:

```
sage: g = Graph()
sage: g.clique_maximum()
[]
```

5.3 Independent sets

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.

5.3.1 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 Wikipedia article Independent_set_(graph_theory).

INPUT:

- `G` – a graph
- `maximal` (boolean) – whether to only consider (inclusionwise) maximal independent sets. Set to `False` by default.
- `complement` (boolean) – whether to consider the graph's complement (i.e. cliques instead of independent sets). Set to `False` by default.

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: print number_of
[1, 10, 30, 30, 5, 0, 0, 0, 0, 0]
```

It is also possible to define an 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()

Computes and returns the number of independent sets

TESTS:

```
sage: from sage.graphs.independent_sets import IndependentSets
sage: IndependentSets(graphs.PetersenGraph()).cardinality()
76
```

Only maximal ones:

```
sage: from sage.graphs.independent_sets import IndependentSets
sage: IndependentSets(graphs.PetersenGraph(), maximal = True).cardinality()
15
```

5.4 Comparability and permutation graphs

Comparability and permutation graphs

This module implements method related to Comparability graphs and Permutation graphs, that is, for the moment, only recognition algorithms.

Most of the information found here can alo be found in [Cleanup] or [ATGA].

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_with_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

5.4.1 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 comparablity graphs, see the corresponding wikipedia page

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 corresponding wikipedia page.

5.4.2 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 [Cleanup]):

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.

5.4.3 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).

5.4.4 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.

5.4.5 References

5.4.6 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`.

EXAMPLE:

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)
(False, [9, 6, 1, 0, 4, 9])
```

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.

EXAMPLE:

The 5-cycle or the Petersen Graph are not transitively orientable:

```
sage: from sage.graphs.comparability import greedy_is_comparability_with_certificate as is_compa
sage: is_comparability(graphs.CycleGraph(5), certificate = True)
(False, [3, 4, 0, 1, 2, 3])
sage: g = graphs.PetersenGraph()
sage: is_comparability(g)
False
sage: is_comparability(g, certificate = True)
(False, [9, 6, 1, 0, 4, 9])
```

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*)

Tests whether the graph is a comparability graph

INPUT:

- `algorithm` – chose 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.

EXAMPLE:

```
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)
```

TESTS:

Let us ensure that no exception is raised when we go over all small graphs:

```
sage: from sage.graphs.comparability import is_comparability
sage: [len([g for g in graphs(i) if is_comparability(g, certificate = True)[0]]) for i in range(
[1, 1, 2, 4, 11, 33, 144]
```

sage.graphs.comparability.**is_comparability_MILP**(*g*, *certificate=False*)

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.

EXAMPLE:

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*)

Tests whether the graph is a permutation graph.

For more information on permutation graphs, refer to the documentation of the comparability module.

INPUT:

- algorithm – chose 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.

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.

EXAMPLE:

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")
```

TESTS:

Trying random permutations, first with the greedy algorithm:

```
sage: from sage.graphs.comparability import is_permutation
sage: for i in range(20):
...       p = Permutations(10).random_element()
...       g1 = graphs.PermutationGraph(p)
...       isit, certif = is_permutation(g1, certificate = True)
...       if not isit:
...          print "Something is wrong here !!"
...          break
...       g2 = graphs.PermutationGraph(*certif)
...       if not g1.is_isomorphic(g2):
...          print "Something is wrong here !!"
...          break
```

Then with MILP:

```
sage: from sage.graphs.comparability import is_permutation
sage: for i in range(20):
```

```
...       p = Permutations(10).random_element()
...       g1 = graphs.PermutationGraph(p)
...       isit, certif = is_permutation(g1, algorithm = "MILP", certificate = True)
...       if not isit:
...          print "Something is wrong here !!"
...          break
...       g2 = graphs.PermutationGraph(*certif)
...       if not g1.is_isomorphic(g2):
...          print "Something is wrong here !!"
...          break
```

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 $uv \notin G$.

EXAMPLE:

```
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: digraphs.DeBruijn(5,2).is_transitive()
False
sage: digraphs.DeBruijn(5,2).is_transitive(certificate = True)
('00', '10')
sage: digraphs.RandomDirectedGNP(20,.2).transitive_closure().is_transitive()
True
```

5.5 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.

5.5.1 Definition

Given a graph G, the *line graph* $L(G)$ of G is the graph such that

$$\begin{aligned} V(L(G)) =& E(G) \\ E(L(G)) =& \{(e, e') : \text{ and } e, e' \text{ have a common endpoint in } G\} \end{aligned}$$

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 page on line graphs.

5.5.2 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 ([Whitney32], [Harary69]), 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 [Beineke70] 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.

5.5.3 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.

Todo

This method sequentially tests each of the forbidden subgraphs in order to know whether the graph is a line graph, which is a very slow method. It could eventually be replaced by `root_graph()` when this method will not require an exponential time to run on general graphs anymore (see its documentation for more information on this problem)... and if it can be improved to return negative certificates !

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)}
```

TESTS:

Disconnected graphs:

```
sage: g = 2*graphs.CycleGraph(3)
sage: gl = g.line_graph().relabel(inplace = False)
sage: new_g = gl.is_line_graph(certificate = True)[1]
sage: g.line_graph().is_isomorphic(gl)
True
```

sage.graphs.line_graph.**line_graph**(*self*, *labels=True*)

Returns the line graph of the (di)graph.

INPUT:

- `labels` (boolean) – 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.

  This is set to `True` by default.

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)]
```

Tests:

trac ticket #13787:

```
sage: g = graphs.KneserGraph(7,1)
sage: C = graphs.CompleteGraph(7)
sage: C.is_isomorphic(g)
True
sage: C.line_graph().is_isomorphic(g.line_graph())
True
```

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) – 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.
- If the graph is *not* a line graph, this implementation will take a looooooong time to run. Its first step is to enumerate all maximal cliques, and that can take a while for general graphs. As soon as there is a way to iterate over maximal cliques without first building the (long) list of them this implementation can be updated, and will deal reasonably with non-line graphs too !

TESTS:

All connected graphs on 6 vertices:

```
sage: from sage.graphs.line_graph import root_graph
sage: def test(g):
...       gl = g.line_graph(labels = False)
...       d=root_graph(gl)
sage: for i,g in enumerate(graphs(6)): # long time
...     if not g.is_connected():       # long time
...       continue                     # long time
...     test(g)                        # long time
```

Non line-graphs:

```
sage: root_graph(graphs.PetersenGraph())
Traceback (most recent call last):
...
ValueError: This graph is not a line graph !
```

Small corner-cases:

```
sage: from sage.graphs.line_graph import root_graph
sage: root_graph(graphs.CompleteGraph(3))
(Complete bipartite graph: Graph on 4 vertices, {0: (0, 1), 1: (0, 2), 2: (0, 3)})
sage: root_graph(graphs.OctahedralGraph())
(Complete graph: Graph on 4 vertices, {0: (0, 1), 1: (0, 2), 2: (0, 3), 3: (1, 2), 4: (1, 3), 5:
sage: root_graph(graphs.DiamondGraph())
(Graph on 4 vertices, {0: (0, 3), 1: (0, 1), 2: (0, 2), 3: (1, 2)})
sage: root_graph(graphs.WheelGraph(5))
(Diamond Graph: Graph on 4 vertices, {0: (1, 2), 1: (0, 1), 2: (0, 2), 3: (2, 3), 4: (1, 3)})
```

5.6 Spanning trees

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 [JoynerNguyenCohen2010] for descriptions of spanning tree algorithms, including minimum spanning trees.

See Also:

- `GenericGraph.min_spanning_tree`.

Todo

- Rewrite `kruskal()` to use priority queues. Once Cython has support for generators and the `yield` statement, rewrite `kruskal()` to use `yield`.
- Prim's algorithm.

- Boruvka's algorithm.
- Parallel version of Boruvka's algorithm.
- Randomized spanning tree construction.

REFERENCES:

5.6.1 Methods

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 simple graphs. Such graphs can be weighted or unweighted.

INPUT:

- `G` – A graph. This can be an undirected graph, a digraph, a multigraph, or a multidigraph. Note the following behaviours:
  - If `G` is unweighted, then consider the simple version of `G` with all self-loops and multiple edges removed.
  - If `G` is directed, then we only consider its undirected version.
  - If `G` is weighted, we ignore all of its self-loops. Note that a weighted graph should only have numeric weights. You cannot assign numeric weights to some edges of `G`, but have `None` as a weight for some other edge. If your input graph is weighted, you are responsible for assign numeric weight to each of its edges. Furthermore, we remove multiple edges as follows. First we convert `G` to be undirected. Suppose there are multiple edges from u to v. Among all such multiple edges, we choose one with minimum weight.
- `wfunction` – A weight function: a function that takes an edge and returns a numeric weight. Default: `None`. The default is to assign each edge a weight of 1.
- `check` – 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?
  - Is `G` directed?
  - Does `G` have self-loops?
  - Does `G` have multiple edges?
  - Is `G` weighted?

  By default, we turn off the sanity checks for performance reasons. This means that by default the function assumes that its input graph is simple, connected, is not a tree, and has at least one vertex. If the input graph does not satisfy all of the latter conditions, you should set `check=True` to perform some sanity checks and preprocessing on the input graph. 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.

- If G is a tree, return the edges of G regardless of whether G is weighted or unweighted, directed or undirected.
- If G is unweighted, default to using unit weight for each edge of G. The default behaviour is to use the already assigned weights of G provided that G is weighted.
- If G is weighted and a weight function is also supplied, then use the already assigned weights of G, not the weight function. If you really want to use a weight function for G even if G is weighted, first convert G to be unweighted and pass in the weight function.

See Also:

- sage.graphs.generic_graph.GenericGraph.min_spanning_tree()

EXAMPLES:

An example from pages 727–728 in [Sahni2000].

```
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: H = DiGraph(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)
...       E = {}
...       for u, v, _ in G.multiple_edges():
...           E.setdefault(u, v)
...       for u in E:
...           W = sorted(G.edge_label(u, E[u]))
...           for w in W[1:]:
...               G.delete_edge(u, E[u], w)
...       G.allow_multiple_edges(False)
sage: sanitize(H)
sage: H
Graph on 7 vertices
sage: kruskal(G, check=True) == kruskal(H, check=True)
True
```

Note that we only consider an undirected version of the input graph. Thus if G is a weighted multidigraph and H is an undirected version of G, then this function should return the same minimum spanning tree for both G and H.

```
sage: from sage.graphs.spanning_tree import kruskal
sage: G = DiGraph({1:{2:[1,14,28], 6:[10]}, 2:{3:[16], 1:[15], 7:[14], 5:[20,21]}, 3:{4:[12,11]}
sage: G.multiple_edges(to_undirected=False)
[(1, 2, 1), (1, 2, 14), (1, 2, 28), (5, 2, 1), (5, 2, 3), (4, 3, 3), (4, 3, 13), (3, 4, 11), (3,
sage: H = G.to_undirected()
sage: H.multiple_edges(to_undirected=True)
[(1, 2, 1), (1, 2, 14), (1, 2, 15), (1, 2, 28), (2, 5, 1), (2, 5, 3), (2, 5, 20), (2, 5, 21), (3
sage: kruskal(G, check=True)
[(1, 2, 1), (1, 6, 10), (2, 3, 16), (2, 5, 1), (2, 7, 14), (3, 4, 3)]
sage: kruskal(G, check=True) == kruskal(H, check=True)
True
sage: G.weighted(True)
sage: H.weighted(True)
sage: kruskal(G, check=True)
[(1, 2, 1), (2, 5, 1), (3, 4, 3), (1, 6, 10), (2, 7, 14), (2, 3, 16)]
sage: kruskal(G, check=True) == kruskal(H, check=True)
True
```

An example from pages 599–601 in [GoodrichTamassia2001].

```
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', 'OR
```

An example from pages 568–569 in [CormenEtAl2001].

```
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: kruskal(G, check=True)
[('g', 'h', 1), ('c', 'i', 2), ('f', 'g', 2), ('a', 'b', 4), ('c', 'f', 4), ('c', 'd', 7), ('a',
```

TESTS:

The input graph must not be empty.

```
sage: from sage.graphs.spanning_tree import kruskal
sage: kruskal(graphs.EmptyGraph(), check=True)
[]
sage: kruskal(Graph(), check=True)
[]
sage: kruskal(Graph(multiedges=True), check=True)
[]
sage: kruskal(Graph(loops=True), check=True)
[]
sage: kruskal(Graph(multiedges=True, loops=True), check=True)
[]
sage: kruskal(DiGraph(), check=True)
```

```
[]
sage: kruskal(DiGraph(multiedges=True), check=True)
[]
sage: kruskal(DiGraph(loops=True), check=True)
[]
sage: kruskal(DiGraph(multiedges=True, loops=True), check=True)
[]
```

The input graph must be connected.

```
sage: def my_disconnected_graph(n, ntries, directed=False, multiedges=False, loops=False):
...       G = Graph()
...       k = randint(1, n)
...       G.add_vertices(range(k))
...       if directed:
...           G = G.to_directed()
...       if multiedges:
...           G.allow_multiple_edges(True)
...       if loops:
...           G.allow_loops(True)
...       for i in range(ntries):
...           u = randint(0, k-1)
...           v = randint(0, k-1)
...           G.add_edge(u, v)
...       while G.is_connected():
...           u = randint(0, k-1)
...           v = randint(0, k-1)
...           G.delete_edge(u, v)
...       return G
sage: G = my_disconnected_graph(100, 50, directed=False, multiedges=False, loops=False)  # long
sage: kruskal(G, check=True)  # long time
[]
sage: G = my_disconnected_graph(100, 50, directed=False, multiedges=True, loops=False)  # long t
sage: kruskal(G, check=True)  # long time
[]
sage: G = my_disconnected_graph(100, 50, directed=False, multiedges=True, loops=True)  # long ti
sage: kruskal(G, check=True)  # long time
[]
sage: G = my_disconnected_graph(100, 50, directed=True, multiedges=False, loops=False)  # long t
sage: kruskal(G, check=True)  # long time
[]
sage: G = my_disconnected_graph(100, 50, directed=True, multiedges=True, loops=False)  # long ti
sage: kruskal(G, check=True)  # long time
[]
sage: G = my_disconnected_graph(100, 50, directed=True, multiedges=True, loops=True)  # long tim
sage: kruskal(G, check=True)  # long time
[]
```

If the input graph is a tree, then return its edges.

```
sage: T = graphs.RandomTree(randint(1, 50))  # long time
sage: T.edges() == kruskal(T, check=True)  # long time
True
```

5.7 PQ-Trees

This module implements PQ-Trees and methods to help recognise Interval Graphs. It is used by `is_interval`.

Author:

- Nathann Cohen

class sage.graphs.pq_trees.**P**(*seq*)
Bases: sage.graphs.pq_trees.PQ

A P-Tree is a PQ-Tree whose children are not ordered (they can be permuted in any way)

set_contiguous(*v*)
Updates `self` so that its sets containing `v` are contiguous for any admissible permutation of its subtrees.

This function also ensures, whenever possible, that all the sets containing `v` are located on an interval on the right side of the ordering.

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.

EXAMPLE:

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*)

This class implements the PQ-Tree, used for the recognition of Interval Graphs, or equivalently for matrices having the so-caled "consecutive ones property".

Briefly, we are given a collection $C = S_1, ..., S_n$ of sets on a common ground set X, and we would like to reorder the elements of C in such a way that for every element $x \in X$ such that $x \in S_i$ and $x \in S_j$, $i < j$, we have $x \in S_l$ for all $i < l < j$. This property could also be rephrased as : the sets containing x are an interval.

To achieve it, we will actually compute ALL the orderings satisfying such constraints using the structure of PQ-Tree, by adding the constraints one at a time.

- At first, there is no constraint : all the permutations are allowed. We will then build a tree composed of one node linked to all the sets in our collection (his children). As we want to remember that all the permutations of his children are allowed, we will label it with "P", making it a P-Tree.
- We are now picking an element $x \in X$, and we want to ensure that all the elements C_x containing it are contiguous. We can remove them from their tree T_1, create a second tree T_2 whose only children are the C_x, and attach this T_2 to T_1. We also make this new tree a $P - Tree$, as all the elements of C_x can be permuted as long as they stay close to each other. Obviously, the whole tree T_2 can be prmuter with the other children of T_1 in any way – it does not impair the fact that the sequence of the children will ensure the sets containing x are contiguous.
- We would like to repeat the same procedure for $x' \in X$, but we are now encountering a problem : there may be sets containing both x' and x, along with others containing only x or only x'. We can permute the sets containing only x' together, or the sets containing both x and x' together, but we may NOT permute all the sets containing x' together as this may break the relationship between the sets containing x. We need Q-Trees. A Q-Tree is a tree whose children are ordered, even though their order could be reversed (if the children of a Q-Tree are $c_1c_2...c_k$, we can change it to $c_k...c_2c_1$). We can now express all the orderings satisfying our two constraints the following way :
  - We create a tree T_1 gathering all the elements not containing x nor x', and make it a $P - Tree$
  - We create 3 P-Trees $T_{x,x'}, T_x, T_{x'}$, which respectively have for children the elements or our collection containing
    - both x and x'
    - only x
    - only x'
  - To ensure our constraints on both elements, we create a Q-tree T_2 whose children are in order $T_x, T_{x,x'}, T_{x'}$
  - We now make this Q-Tree T_2 a children of the P-Tree T_1

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 or constraints, or to prove that no such ordering exists. This is the whole purpose of this class and this algorithm, and is explained with more details in many places, for example in the following document from Hajiaghayi [Haj].

REFERENCES:

AUTHOR : Nathann Cohen

cardinality()

Returns the number of children of `self`

EXAMPLE:

```
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.cardinality()
3
```

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.

EXAMPLE:

```
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}])
```

is_P()

Tests whether `self` is a P-Tree

EXAMPLE:

```
sage: from sage.graphs.pq_trees import P, Q
sage: P([[0,1],[2,3]]).is_P()
True
sage: Q([[0,1],[2,3]]).is_P()
False
```

is_Q()

Tests whether `self` is a Q-Tree

EXAMPLE:

```
sage: from sage.graphs.pq_trees import P, Q
sage: Q([[0,1],[2,3]]).is_Q()
True
sage: P([[0,1],[2,3]]).is_Q()
False
```

ordering()

Returns the current ordering given by listing the leaves from left to right.

EXAMPLE:

```
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

EXAMPLE:

```
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 (booleans) – 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}]
```

split(*v*)

Returns the subsequences of children containing and not containing v

INPUT:

- v – an element of the ground set

OUTPUT:

Two lists, the first containing the children of self containing v, and the other containing the other children.

Note: This command is meant to be used on a partial tree, once it has be "set continuous" on an element v and aligned it to the right. Hence, none of the list should be empty (an exception is raised if that happens, as it would reveal a bug in the algorithm) and the sum contains + does_not_contain should be equal to the sequence of children of self.

EXAMPLE:

```
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.reverse()
```

```
sage: contains, does_not_contain = p.split(1)
sage: contains
[{1, 2}]
sage: does_not_contain
[('P', [{9, 2}, {8, 2}, {2, 4}]), {2, 3}]
sage: does_not_contain + contains == p._children
True
```

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

set_contiguous(*v*)

Updates `self` so that its sets containing `v` are contiguous for any admissible permutation of its subtrees.

This function also ensures, whenever possible, that all the sets containing `v` are located on an interval on the right side of the ordering.

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.

EXAMPLE:

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*)

x.__init__(...) initializes x; see help(type(x)) for signature

sage.graphs.pq_trees.**new_P**(*liste*)
x.__init__(...) initializes x; see help(type(x)) for signature

sage.graphs.pq_trees.**new_Q**(*liste*)
x.__init__(...) initializes x; see help(type(x)) for signature

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

EXAMPLE:

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,
```

sage.graphs.pq_trees.**set_contiguous**(*tree*, *x*)
x.__init__(...) initializes x; see help(type(x)) for signature

5.8 Generation of trees

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 [WRIGHT-ETAL].

AUTHORS:

- Ryan Dingman (2009-04-16): initial version

REFERENCES:

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 t.is_tree() == False:
...               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) == True:
...                   return False
...           trees.append(t)
...       return True
sage: print check_trees(10)
True

sage: from sage.graphs.trees import TreeIterator
sage: count = 0
sage: for t in TreeIterator(15):
...       count += 1
sage: print count
7741
```

next()
x.next() -> the next value, or raise StopIteration

5.9 Matching Polynomial

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:

5.9.1 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
```

TESTS:

Checking the numerical results up to 20:

```
sage: from sage.functions.orthogonal_polys import hermite
sage: p = lambda n: 2^(-n/2)*hermite(n, x/sqrt(2))
sage: all(p(i) == complete_poly(i) for i in range(2, 20))
True
```

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 [Godsil93]:

$$\mu(x) = \sum_{k \geq 0} (-1)^k p(G, k) x^{n-2k}$$

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 [Godsil93]:

- 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 [Godsil93] 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
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))
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)
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
```

TESTS:

Non-integer labels should work, (trac ticket #15545):

```
sage: G = Graph(10);
sage: G.add_vertex((0,1))
sage: G.add_vertex('X')
sage: G.matching_polynomial()
x^12
```

5.10 Genus

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, implementation='c_graph', sparse=False)
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
sage: bt.genus() #long time
1
sage: bt.genus(cutoff=1)
1
sage: G = graphs.PetersenGraph()
sage: G = Graph(G, implementation='c_graph', sparse=False)
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
sage: bt.genus()
1
sage: G = graphs.FlowerSnark()
sage: G = Graph(G, implementation='c_graph', sparse=False)
sage: bt = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
sage: bt.genus()
2
```

genus (*style=1*, *cutoff=0*, *record_embedding=0*)

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` – int, find minimum genus if 1, maximum genus if 2
- `cutoff` – int, stop searching if search style is 1 and genus <= cutoff, or if style is 2 and genus >= cutoff. This is useful where the genus of the graph has a known bound.

- record_embedding – bool, 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), implementation='c_graph', sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
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(implementation='c_graph', sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
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.

DOCTESTS:

```
sage: import sage.graphs.genus
sage: G = Graph(graphs.CompleteGraph(5), implementation='c_graph', sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
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(implementation='c_graph', sparse=False)
sage: G.add_edge(0,1)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
sage: gb.get_embedding()
{0: [1], 1: [0]}
sage: G = Graph(implementation='c_graph', sparse=False)
sage: gb = sage.graphs.genus.simple_connected_genus_backtracker(G._backend._cg)
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
sage: G = graphs.OddGraph(3)
sage: genus(G)
1
```

REFERENCS:

[1] http://www.springerlink.com/content/0776127h0r7548v7/

5.11 Linear Extensions of Directed Acyclic Graphs.

A linear extension of a directed acyclic graph is a total (linear) ordering on the vertices that is compatible with the graph in the following sense: if there is a path from x to y in the graph, the x appears before y in the linear extension.

The algorithm implemented in this module is from "Generating Linear Extensions Fast" by Preusse and Ruskey, which can be found at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.3057 . The algorithm generates the extensions in constant amortized time (CAT) – a constant amount of time per extension generated, or linear in the number of extensions generated.

EXAMPLES:

Here we generate the 5 linear extensions of the following directed acyclic graph:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.is_directed_acyclic()
True
sage: LinearExtensions(D).list()
[[0, 1, 2, 3, 4],
 [0, 1, 2, 4, 3],
 [0, 2, 1, 3, 4],
 [0, 2, 1, 4, 3],
 [0, 2, 4, 1, 3]]
```

Notice how all of the total orders are compatible with the ordering induced from the graph.

We can also get at the linear extensions directly from the graph. From the graph, the linear extensions are known as topological sorts

```
sage: D.topological_sort_generator()
[[0, 1, 2, 3, 4],
 [0, 1, 2, 4, 3],
 [0, 2, 1, 3, 4],
```

```
[0, 2, 1, 4, 3],
[0, 2, 4, 1, 3]]
```

class sage.graphs.linearextensions.**LinearExtensions**(*dag*)
Bases: sage.combinat.combinat.CombinatorialClass

Creates an object representing the class of all linear extensions of the directed acyclic graph code{dag}.

Note that upon construction of this object some pre-computation is done. This is the "preprocessing routine" found in Figure 7 of "Generating Linear Extensions Fast" by Preusse and Ruskey.

This is an in-place algorithm and the list self.le keeps track of the current linear extensions. The boolean variable self.is_plus keeps track of the "sign".

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: l == loads(dumps(l))
True
```

generate_linear_extensions(*i*)
This a Python version of the GenLE routine found in Figure 8 of "Generating Linear Extensions Fast" by Pruesse and Ruskey.

Note that this is meant to be called by the list method and is not meant to be used directly.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: l.linear_extensions = []
sage: l.linear_extensions.append(l.le[:])
sage: l.generate_linear_extensions(l.max_pair)
sage: l.linear_extensions
[[0, 1, 2, 3, 4], [0, 2, 1, 3, 4]]
```

incomparable(*x*, *y*)
Returns True if vertices x and y are incomparable in the directed acyclic graph when thought of as a poset.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: l.incomparable(0,1)
False
sage: l.incomparable(1,2)
True
```

list()
Returns a list of the linear extensions of the directed acyclic graph.

Note that once they are computed, the linear extensions are cached in this object.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: LinearExtensions(D).list()
```

```
[[0, 1, 2, 3, 4],
 [0, 1, 2, 4, 3],
 [0, 2, 1, 3, 4],
 [0, 2, 1, 4, 3],
 [0, 2, 4, 1, 3]]
```

move(*element*, *direction*)

This implements the Move procedure described on page 7 of "Generating Linear Extensions Fast" by Pruesse and Ruskey.

If direction is "left", then this transposes element with the element on its left. If the direction is "right", then this transposes element with the element on its right.

Note that this is meant to be called by the generate_linear_extensions method and is not meant to be used directly.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: _ = l.list()
sage: l.le = [0, 1, 2, 3, 4]
sage: l.move(1, "left")
sage: l.le
[1, 0, 2, 3, 4]
sage: l.move(1, "right")
sage: l.le
[0, 1, 2, 3, 4]
```

right(*i*, *letter*)

If letter ==”b”, then this returns True if and only if self.b[i] is incomparable with the elements to its right in self.le. If letter == "a", then it returns True if and only if self.a[i] is incomparable with the element to its right in self.le and the element to the right is not self.b[i]

This is the Right function described on page 8 of "Generating Linear Extensions Fast" by Pruesse and Ruskey.

Note that this is meant to be called by the generate_linear_extensions method and is not meant to be used directly.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: _ = l.list()
sage: l.le
[0, 1, 2, 4, 3]
sage: l.a
[1, 4]
sage: l.b
[2, 3]
sage: l.right(0, "a")
False
sage: l.right(1, "a")
False
sage: l.right(0, "b")
False
sage: l.right(1, "b")
```

```
False
```

switch(*i*)

This implements the Switch procedure described on page 7 of "Generating Linear Extensions Fast" by Pruesse and Ruskey.

If i == -1, then the sign is changed. If i > 0, then self.a[i] and self.b[i] are transposed.

Note that this meant to be called by the generate_linear_extensions method and is not meant to be used directly.

EXAMPLES:

```
sage: from sage.graphs.linearextensions import LinearExtensions
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: l = LinearExtensions(D)
sage: _ = l.list()
sage: l.le = [0, 1, 2, 3, 4]
sage: l.is_plus
True
sage: l.switch(-1)
sage: l.is_plus
False
sage: l.a
[1, 4]
sage: l.b
[2, 3]
sage: l.switch(0)
sage: l.le
[0, 2, 1, 3, 4]
sage: l.a
[2, 4]
sage: l.b
[1, 3]
```

5.12 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.

AUTHORS: – Jonathan Bober, Emily Kirkman (2008-02-09): initial version

REFERENCE:

[1] Schnyder, Walter. Embedding Planar Graphs on the Grid. Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco (1994), pp. 138-147.

class sage.graphs.schnyder.**TreeNode**(*parent=None*, *children=None*, *label=None*)

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
```

5.13 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:

```
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
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 the octahedral, dodecahedral, random and empty 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`.

partition	A partition of the vertex set. If specified, plot will show each cell in a different color. vertex_colors takes precedence.
dist	The distance between multiedges.
vertex_labels	Whether or not to draw vertex labels.
edge_color	The default color for edges.
spring	Use spring layout to finalize the current layout.
pos	The position dictionary of vertices
loop_size	The radius of the smallest loop.
color_by_label	Whether to color the edges according to their labels. This also accepts a function or dictionary mapping labels to colors.
iterations	The number of times to execute the spring layout algorithm.
talk	Whether to display the vertices in talk mode (larger and white).
edge_labels	Whether or not to draw edge labels.
vertex_size	The size to draw the vertices.
dim	The dimension of the layout – 2 or 3.
edge_style	The linestyle of the edges. It should be one of "solid", "dashed", "dotted", dashdot", or "-", "–", ":", "-.", respectively. This currently only works for directed graphs, since we pass off the undirected graph to networkx.
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).
vertex_shape	The shape to draw the vertices. Currently unavailable for Multi-edged DiGraphs.
vertex_colors	Dictionary of vertex coloring : 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 does not get drawn).
by_component	Whether to do the spring layout by connected component – a boolean.
heights	A dictionary mapping heights to the list of vertices at this height.
graph_border	Whether or not to draw a frame around the graph.
max_dist	The max distance range to allow multiedges.
prog	Which graphviz layout program to use – one of "circo", "dot", "fdp", "neato", or "twopi".
edge_colors	a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges.
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.
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'`

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: 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()`	Sets the position plotting parameters for this GraphPlot.
`GraphPlot.set_vertices()`	Sets the vertex plotting parameters for this GraphPlot.
`GraphPlot.set_edges()`	Sets the edge (or arrow) plotting parameters for the GraphPlot object.
`GraphPlot.show()`	Shows the (Di)Graph associated with this GraphPlot object.
`GraphPlot.plot()`	Returns a graphics object representing the (di)graph.
`GraphPlot.layout_tree()`	Compute a nice layout of a tree.
`_circle_embedding()`	Sets some vertices on a circle in the embedding of a graph G.
`_line_embedding()`	Sets some vertices on a line in the embedding of a graph G.

5.13.1 Methods and classes

`sage.graphs.graph_plot.`**`_circle_embedding`**(*g*, *vertices*, *center=(0, 0)*, *radius=1*, *shift=0*)

Sets some vertices on a circle in the embedding of a graph G.

This method modifies the graph's embedding so that the vertices listed in `vertices` appear in this ordering on a circle of given radius and center. The `shift` parameter is actually a 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).

EXAMPLE:

```
sage: from sage.graphs.graph_plot import _circle_embedding
sage: g = graphs.CycleGraph(5)
sage: _circle_embedding(g, [0, 2, 4, 1, 3], radius=2, shift=.5)
sage: g.show()
```

`sage.graphs.graph_plot.`**`_line_embedding`**(*g*, *vertices*, *first=(0, 0)*, *last=(0, 1)*)

Sets some vertices on a line in the embedding of a graph G.

This method modifies the graph's embedding so that the vertices of `vertices` appear on a line, where the position of `vertices[0]` is the pair `first` and the position of `vertices[-1]` is `last`. The vertices are evenly spaced.

EXAMPLE:

```
sage: from sage.graphs.graph_plot import _line_embedding
sage: g = graphs.PathGraph(5)
sage: _line_embedding(g, [0, 2, 4, 1, 3], first=(-1, -1), last=(1, 1))
sage: g.show()
```

class sage.graphs.graph_plot.**GraphPlot**(*graph*, *options*)
Bases: sage.structure.sage_object.SageObject

Returns 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.

EXAMPLE:

```
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,
...       'graph_border':False,
...       'talk':False,
...       'color_by_label':False,
...       'partition':None,
...       'dist':.075,
...       'max_dist':1.5,
...       'loop_size':.075}
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: T = graphs.RandomLobster(25,0.3,0.3)
sage: T.show(layout='tree',tree_orientation='up') # indirect doctest
```

```
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*)
Returns 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,
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(10)[5:]:
...     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.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
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()
```

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)
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()

sage: G = Graph()
sage: P = G.graphplot().plot()
sage: P.axes()
False
sage: G = DiGraph()
sage: P = G.graphplot().plot()
sage: P.axes()
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()

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()
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()

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()

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()
```

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()
```

TESTS:

Make sure that show options work with plot also:

```
sage: g = Graph({})
sage: g.plot(title='empty graph', axes=True)
```

Check for invalid inputs:

```
sage: p = graphs.PetersenGraph().plot(egabrag='garbage')
Traceback (most recent call last):
...
ValueError: Invalid input 'egabrag=garbage'
```

set_edges(***edge_options*)

Sets 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, edge_style='d
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_edges(edge_color='black')
sage: GP.plot()

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='d
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_edges(edge_color='black')
sage: GP.plot()
```

TESTS:

```
sage: G = Graph("Fooba")
sage: G.show(edge_colors={'red':[(3,6),(2,5)]})
```

Verify that default edge labels are pretty close to being between the vertices in some cases where they weren't due to truncating division (trac ticket #10124):

```
sage: test_graphs = graphs.FruchtGraph(), graphs.BullGraph()
sage: tol = 0.001
sage: for G in test_graphs:
...       E=G.edges()
...       for e0, e1, elab in E:
...           G.set_edge_label(e0, e1, '%d %d' % (e0, e1))
...       gp = G.graphplot(save_pos=True,edge_labels=True)
...       vx = gp._plot_components['vertices'][0].xdata
...       vy = gp._plot_components['vertices'][0].ydata
...       for elab in gp._plot_components['edge_labels']:
...           textobj = elab[0]
...           x, y, s = textobj.x, textobj.y, textobj.string
...           v0, v1 = map(int, s.split())
...           vn = vector(((x-(vx[v0]+vx[v1])/2.),y-(vy[v0]+vy[v1])/2.)).norm()
...           assert vn < tol
```

set_pos()

Sets 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: [...e-17, 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]})
```

TESTS:

Make sure that vertex locations are floats. Not being floats isn't a bug in itself but makes it too easy to accidentally introduce a bug elsewhere, such as in `set_edges()` (trac ticket #10124), via silent truncating division of integers:

```
sage: g = graphs.FruchtGraph()
sage: gp = g.graphplot()
sage: set(map(type, flatten(gp._pos.values())))
set([<type 'float'>])
sage: g = graphs.BullGraph()
sage: gp = g.graphplot(save_pos=True)
sage: set(map(type, flatten(gp._pos.values())))
set([<type 'float'>])
```

set_vertices(***vertex_options*)

Sets 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='d
sage: GP.set_vertices(talk=True)
sage: GP.plot()
sage: GP.set_vertices(vertex_colors='pink', vertex_shape='^')
sage: GP.plot()
```

show(***kwds*)

Shows the (Di)Graph associated with this `GraphPlot` object.

INPUT:

This method accepts all parameters of `sage.plot.graphics.Graphics.show()`.

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.

EXAMPLE:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

5.14 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 dictionay 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 with the Sage command `install_package('d3js')` or 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.

5.14.1 Functions

`sage.graphs.graph_plot_js.`**`gen_html_code`**(*G*, *vertex_labels=False*, *edge_labels=False*, *vertex_partition=[]*, *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) – Whether to display vertex labels (set to `False` by default).
- `edge_labels` (boolean) – Whether to display edge labels (set to `False` by default).
- `vertex_partition` – a list of lists representing a partition of the vertex set. Vertices are then colored in the graph according to the partition. Set to `[]` by default.
- `edge_partition` – same as `vertex_partition`, with edges instead. Set to `[]` by default.
- `force_spring_layout` – whether to take sage's position into account if there is one (see `()` and `()`), or to compute a spring layout. Set to `False` by default.
- `vertex_size` – The size of a vertex' circle. Set to 7 by default.
- `edge_thickness` – Thickness of an edge. Set to `4` by default.
- `charge` – the vertices' charge. Defines how they repulse each other. See https://github.com/mbostock/d3/wiki/Force-Layout for more information. Set to `-120` by default.

- `link_distance` – See https://github.com/mbostock/d3/wiki/Force-Layout for more information. Set to `30` by default.
- `link_strength` – See https://github.com/mbostock/d3/wiki/Force-Layout for more information. Set to `2` by default.
- `gravity` – See https://github.com/mbostock/d3/wiki/Force-Layout for more information. Set to `0.04` by default.

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 with the Sage command `install_package('d3js')` or 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 -- inter

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
```

TESTS:

```
sage: from sage.graphs.graph_plot_js import gen_html_code
sage: filename = gen_html_code(graphs.PetersenGraph())
```

5.15 Vertex separation

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 [Kin92] and Bodlaender [Bod98].

This module contains the following methods

`path_decomposition()`	Returns the pathwidth of the given graph and the ordering of the vertices resulting in a corresponding path decomposition
`vertex_separation()`	Returns an optimal ordering of the vertices and its cost for vertex-separation
`vertex_separation_MILP()`	Computes the vertex separation of G and the optimal ordering of its vertices using an MILP formulation
`lower_bound()`	Returns 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()`	Returns the width of the path decomposition induced by the linear ordering L of the vertices of G

5.15.1 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 algrithm.

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.

5.15.2 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 [SP10]. 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:

$$\text{Minimize: } z \tag{5.1}$$

$$\text{Such that: } x_v^t \leq x_v^{t+1} \qquad \forall v \in V,\ 0 \leq t \leq n-2 \tag{5.2}$$

$$y_v^t \leq y_v^{t+1} \qquad \forall v \in V,\ 0 \leq t \leq n-2 \tag{5.3}$$

$$y_v^t \leq x_w^t \quad \forall v \in V,\ \forall w \in N^+(v),\ 0 \leq t \leq n-1 \tag{5.4}$$

$$\sum_{v \in V} y_v^t = t+1 \qquad 0 \leq t \leq n-1 \tag{5.5}$$

$$x_v^t - y_v^t \leq u_v^t \qquad \forall v \in V,\ 0 \leq t \leq n-1 \tag{5.6}$$

$$\sum_{v \in V} u_v^t \leq z \qquad 0 \leq t \leq n-1 \tag{5.7}$$

$$0 \leq x_v^t \leq 1 \qquad \forall v \in V,\ 0 \leq t \leq n-1 \tag{5.8}$$

$$0 \leq u_v^t \leq 1 \qquad \forall v \in V,\ 0 \leq t \leq n-1 \tag{5.9}$$

$$y_v^t \in \{0, 1\} \qquad \forall v \in V,\ 0 \leq t \leq n-1 \tag{5.10}$$

$$0 \leq z \leq n \tag{5.11}$$

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 $y_v^t == 1$.

5.15.3 REFERENCES

5.15.4 AUTHORS

- Nathann Cohen (2011-10): Initial version and exact exponential algorithm

- David Coudert (2012-04): MILP formulation and tests functions

5.15.5 METHODS

class sage.graphs.graph_decompositions.vertex_separation.**FastDigraph**
Bases: object

x.__init__(...) initializes x; see help(type(x)) for signature

print_adjacency_matrix()
Displays the adjacency matrix of self.

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 oterwise.

EXAMPLE:

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
```

TEST:

Giving anything else than a Graph or a DiGraph:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: vertex_separation.is_valid_ordering(2, [])
Traceback (most recent call last):
...
ValueError: The input parameter must be a Graph or a DiGraph.
```

Giving anything else than a list:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = graphs.CycleGraph(6)
sage: vertex_separation.is_valid_ordering(G, {})
Traceback (most recent call last):
...
ValueError: The second parameter must be of type 'list'.
```

sage.graphs.graph_decompositions.vertex_separation.**lower_bound**(G)
Returns a lower bound on the vertex separation of G

INPUT:

- G – 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.

EXAMPLE:

On a circuit:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import lower_bound
sage: g = digraphs.Circuit(6)
sage: lower_bound(g)
1
```

TEST:

Given anything else than a DiGraph:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import lower_bound
sage: g = graphs.CycleGraph(5)
sage: lower_bound(g)
Traceback (most recent call last):
...
ValueError: The parameter must be a DiGraph.
```

sage.graphs.graph_decompositions.vertex_separation.**path_decomposition**(*G*, *algorithm='exponential'*, *verbose=False*)

Returns the pathwidth of the given graph and the ordering of the vertices resulting in a corresponding path decomposition.

INPUT:

- G – a digraph
- algorithm – (default: "exponential") Specify the algorithm to use among
  - 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) – 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 exponential 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.

See Also:

- Graph.treewidth() – computes the treewidth of a graph

EXAMPLE:

The vertex separation of a cycle is equal to 2:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import path_decomposition
sage: g = graphs.CycleGraph(6)
sage: pw, L = path_decomposition(g); pw
2
sage: pwm, Lm = path_decomposition(g, algorithm = "MILP"); pwm
2
```

TEST:

Given anything else than a Graph:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import path_decomposition
sage: g = digraphs.Circuit(6)
sage: path_decomposition(g)
Traceback (most recent call last):
...
ValueError: The parameter must be a Graph.
```

sage.graphs.graph_decompositions.vertex_separation.**test_popcount**()

Correction test for popcount32.

EXAMPLE:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import test_popcount
sage: test_popcount() # not tested
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation**(*G*, *verbose=False*)

Returns an optimal ordering of the vertices and its cost for vertex-separation.

INPUT:

- `G` – a digraph
- `verbose` (boolean) – 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.

EXAMPLE:

The vertex separation of a circuit is equal to 1:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_separation
sage: g = digraphs.Circuit(6)
sage: vertex_separation(g)
(1, [0, 1, 2, 3, 4, 5])
```

TEST:

Given anything else than a DiGraph:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import lower_bound
sage: g = graphs.CycleGraph(5)
sage: lower_bound(g)
```

```
Traceback (most recent call last):
...
ValueError: The parameter must be a DiGraph.
```

Graphs with non-integer vertices:

```
sage: from sage.graphs.graph_decompositions.vertex_separation import vertex_separation
sage: D=digraphs.DeBruijn(2,3)
sage: vertex_separation(D)
(2, ['000', '001', '100', '010', '101', '011', '110', '111'])
```

sage.graphs.graph_decompositions.vertex_separation.**vertex_separation_MILP**(*G*, *integrality=False*, *solver=None*, *verbosity=0*)

Computes 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 [SP10]. See the `module's documentation` for more details on this MILP formulation.

INPUTS:

- `G` – a DiGraph
- `integrality` – (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` – (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.

EXAMPLE:

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
```

TESTS:

Comparison with exponential algorithm:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: for i in range(10):
...       G = digraphs.RandomDirectedGNP(10, 0.2)
...       ve, le = vertex_separation.vertex_separation(G)
...       vm, lm = vertex_separation.vertex_separation_MILP(G)
...       if ve != vm:
...          print "The solution is not optimal!"
```

Comparison with different values of the integrality parameter:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: for i in range(10):  # long time (11s on sage.math, 2012)
....:     G = digraphs.RandomDirectedGNP(10, 0.2)
....:     va, la = vertex_separation.vertex_separation_MILP(G, integrality=False)
....:     vb, lb = vertex_separation.vertex_separation_MILP(G, integrality=True)
....:     if va != vb:
....:        print "The integrality parameter changes the result!"
```

Giving anything else than a DiGraph:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: vertex_separation.vertex_separation_MILP([])
Traceback (most recent call last):
...
ValueError: The first input parameter must be a DiGraph.
```

sage.graphs.graph_decompositions.vertex_separation.**width_of_path_decomposition**(*G*, *L*)

Returns 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
```

TESTS:

Path decomposition of a BalancedTree:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = graphs.BalancedTree(3,2)
sage: pw, L = vertex_separation.path_decomposition(G)
sage: pw == vertex_separation.width_of_path_decomposition(G, L)
True
sage: L.reverse()
sage: pw == vertex_separation.width_of_path_decomposition(G, L)
False
```

Directed path decomposition of a circuit:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = digraphs.Circuit(8)
sage: vs, L = vertex_separation.vertex_separation(G)
sage: vs == vertex_separation.width_of_path_decomposition(G, L)
True
sage: L = [0,4,6,3,1,5,2,7]
sage: vs == vertex_separation.width_of_path_decomposition(G, L)
False
```

Giving a wrong linear ordering:

```
sage: from sage.graphs.graph_decompositions import vertex_separation
sage: G = Graph()
sage: vertex_separation.width_of_path_decomposition(G, ['a','b'])
Traceback (most recent call last):
...
ValueError: The input linear vertex ordering L is not valid for G.
```

5.16 Rank Decompositions of graphs

Rank Decompositions of graphs

This modules wraps a C code from Philipp Klaus Krause computing a an optimal rank-decomposition [RWKlause].

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 qual 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 noes have degree 3. In a tree, ay edge naturally corresponds to a bipartition of the vertex set : indeed, the reoal 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 [RWKlause] is a program that calculates rank-width and rank-decompositions. It is based on ideas from :

- "Computing rank-width exactly" by Sang-il Oum [Oum]
- "Sopra una formula numerica" by Ernesto Pascal
- "Generation of a Vector from the Lexicographical Index" by B.P. Buckles and M. Lybanon [BL]
- "Fast additions on masked integers" by Michael D. Adams and David S. Wise [AW]

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 comarable 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 !

EXAMPLE:

```
sage: g = graphs.PetersenGraph()
sage: g.rank_decomposition()
(3, Graph on 19 vertices)
```

AUTHORS:

- Philipp Klaus Krause : Implementation of the C algorithm [RWKlause].
- Nathann Cohen : Interface with Sage and documentation.

REFERENCES:

5.16.1 Methods

sage.graphs.graph_decompositions.rankwidth.**mkgraph**()

Returns the graph corresponding the the current rank-decomposition.

(This function is for internal use)

EXAMPLE:

```
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*)

Computes 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) – 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).

EXAMPLE:

```
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)
```

5.17 Products of graphs

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.

References:

Author:

- Nathann Cohen (May 2012 – coded while watching the election of Francois Hollande on TV)

5.17.1 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 $G = G_1\Box \cdots \Box G_k$?

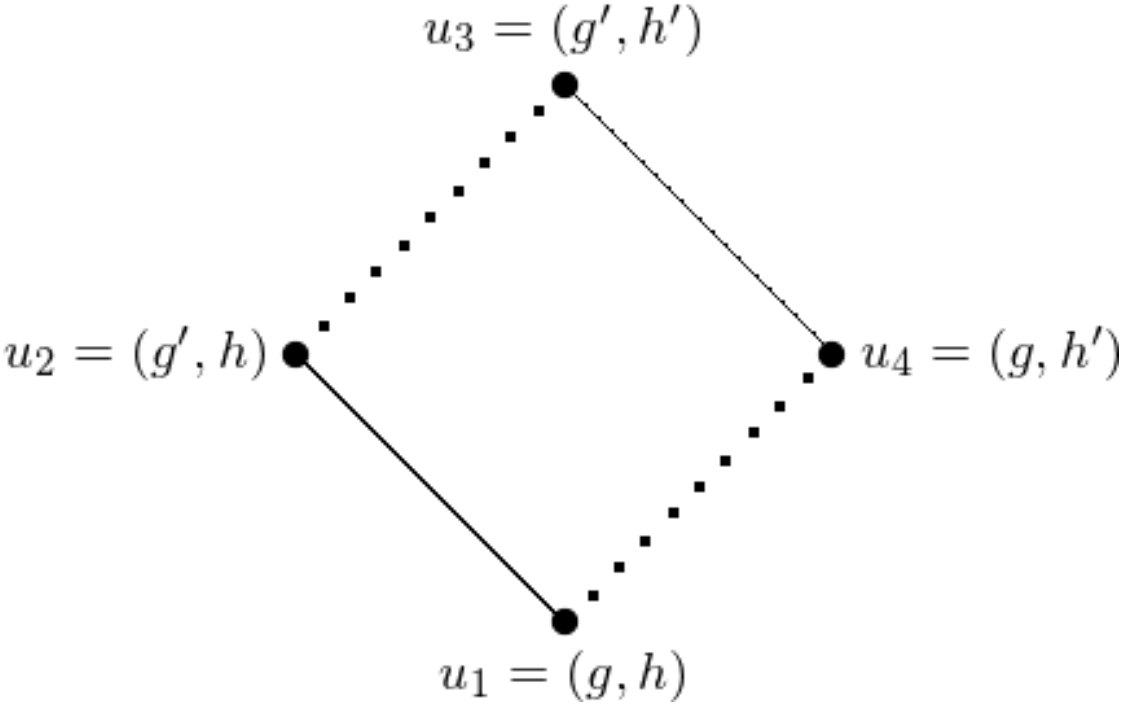
This problem can actually be solved, and the resulting factorization is unique. What is explained below can be found in the book *Handbook of Product Graphs* [HIK11].

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 $G\Box H$ 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 [HIK11] 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.

5.17.2 Methods

sage.graphs.graph_decompositions.graph_products.**is_cartesian_product**(*g*, *certificate=False*, *relabeling=False*)

Tests whether the graph is a cartesian product.

INPUT:

- certificate (boolean) – 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) – 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.

  This is set to False by default.

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 !

EXAMPLE:

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: g.is_cartesian_product(g, relabeling = True)
(True, {0: (0, 0), 1: (0, 1), 2: (0, 2), 3: (0, 3),
        4: (0, 4), 5: (5, 0), 6: (5, 1), 7: (5, 2),
        8: (5, 3), 9: (5, 4), 10: (10, 0), 11: (10, 1),
        12: (10, 2), 13: (10, 3), 14: (10, 4), 15: (15, 0),
        16: (15, 1), 17: (15, 2), 18: (15, 3), 19: (15, 4),
        20: (20, 0), 21: (20, 1), 22: (20, 2), 23: (20, 3),
        24: (20, 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
```

TESTS:

Wagner's Graph (trac ticket #13599):

```
sage: g = graphs.WagnerGraph()
sage: g.is_cartesian_product()
False
```

5.18 Modular decomposition

Modular decomposition

sage.graphs.modular_decomposition.modular_decomposition.**modular_decomposition**(*g*)

Returns a modular decomposition of the given graph.

INPUT:

- g – a graph

OUTPUT:

A pair of two values (recursively encoding the decomposition) :

- The type of the current module :
  - "Parallel"
  - "Prime"
  - "Serie"
- The list of submodules (as list of pairs (type, list), recursively...) or the vertex's name if the module is a singleton.

Note: As this fuction could be used by efficient C routines, the vertices returned are not labels but identifiants from [0, ..., g.order()-1]

ALGORITHM:

This function uses a C implementation of a 2-step algorithm implemented by Fabien de Montgolfier [FMDecb] :

- Computation of a factorizing permutation [HabibViennot1999b].
- Computation of the tree itself [CapHabMont02b].

EXAMPLES:

The Bull Graph is prime:

```
sage: from sage.graphs.modular_decomposition.modular_decomposition import modular_decomposition
sage: modular_decomposition(graphs.BullGraph())
('Prime', [3, 4, 0, 1, 2])
```

The Petersen Graph too:

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

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)
sage: modular_decomposition(g)
('Serie', [0, ('Parallel', [5, ('Serie', [1, 4, 3, 2]), 6])])
```

REFERENCES:

5.19 Convexity properties of graphs

Convexity properties of graphs

This class gathers the algorithms related to convexity in a graph. It implements the following methods:

`ConvexityProperties.hull()`	Returns the convex hull of a set of vertices
`ConvexityProperties.hull_number()`	Computes 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

5.19.1 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-conected 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.

EXAMPLE:

```
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
```

TESTS:

```
sage: ConvexityProperties(digraphs.Circuit(5))
Traceback (most recent call last):
...
ValueError: This is currenly implemented for Graphs only.Only minor updates are needed if you wa
```

hull(*vertices*)

Returns the convex hull of a set of vertices.

INPUT:

- `vertices` – A list of vertices.

EXAMPLE:

```
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*)

Computes 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) – whether to return only the hull number (default) or a minimum set whose convex hull is the whole graph.
- `verbose` (boolean) – whether to display information on the LP.

COMPLEXITY:

This problem is NP-Hard [CHZ02], 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 constrait 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.

EXAMPLE:

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]
```

REFERENCE:

5.20 Weakly chordal graphs

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)

REFERENCES:

5.20.1 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 $O(m^2)$ [NikolopoulosPalios07] (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
```

TESTS:

Further tests:

```
sage: g = Graph({0:[6,7],1:[7,8],2:[8,9],3:[9,10],4:[10,11],5:[11,6],6:[0,5,7],7:[0,1,6],8:[1,2,
sage: r,a = g.is_long_antihole_free(certificate=True)
sage: r
False
sage: a.complement().is_isomorphic( graphs.CycleGraph(9) )
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 $O(m^2)$ [NikolopoulosPalios07] (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
```

TESTS:

Another graph with vertices 2, ..., 8, 10:

```
sage: g = Graph({2:[3,8],3:[2,4],4:[3,8,10],5:[6,10],6:[5,7],7:[6,8],8:[2,4,7,10],10:[4,5,8]})
sage: r,hole = g.is_long_hole_free(certificate=True)
sage: r
False
sage: hole
Subgraph of (): Graph on 5 vertices
sage: hole.is_isomorphic(graphs.CycleGraph(hole.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 = len(s.vertices())
sage: s.is_isomorphic( graphs.CycleGraph(l) )
True
```

5.21 Distances/shortest paths between all pairs of vertices

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.

5.21.1 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()`.
- 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)

REFERENCE:

5.21.2 Functions

sage.graphs.distances_all_pairs.**diameter**(G)

Returns the diameter of G.

EXAMPLE:

```
sage: from sage.graphs.distances_all_pairs import diameter
sage: g = graphs.PetersenGraph()
sage: diameter(g)
2
```

sage.graphs.distances_all_pairs.**distances_all_pairs**(G)

Returns 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]`.

EXAMPLE:

```
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)

Returns 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.

The integer corresponding to a vertex is its index in the list `G.vertices()`.

EXAMPLE:

```
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)

Returns 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*)

Returns the vector of eccentricities in G.

The array returned is of length n, and its ith component is the eccentricity of the ith vertex in `G.vertices()`.

EXAMPLE:

```
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.graphs.distances_all_pairs.**floyd_warshall**(*gg*, *paths=True*, *distances=False*)

Computes the shortest path/distances between all pairs of vertices.

For more information on the Floyd-Warshall algorithm, see the Wikipedia article on Floyd-Warshall.

INPUT:

- `gg` – the graph on which to work.
- `paths` (boolean) – whether to return the dictionary of shortest paths. Set to `True` by default.
- `distances` (boolean) – whether to return the dictionary of distances. Set to `False` by default.

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)
{(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}}
```

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}}
```

TESTS:

Too large graphs:

```
sage: from sage.graphs.distances_all_pairs import floyd_warshall
sage: floyd_warshall(Graph(65536))
Traceback (most recent call last):
...
ValueError: The graph backend contains more than 65535 nodes
```

sage.graphs.distances_all_pairs.**is_distance_regular**(*G*, *parameters=False*)

Tests 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 its associated wikipedia page.

INPUT:

- `parameters` (boolean) – if set to `True`, the function returns the pair `(b,c)` of lists of integers instead of `True` (see the definition above). Set to `False` by default.

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
```

TESTS:

```
sage: graphs.PathGraph(2).is_distance_regular(parameters = True)
([1, None], [None, 1])
sage: graphs.Tutte12Cage().is_distance_regular(parameters=True)
([3, 2, 2, 2, 2, 2, None], [None, 1, 1, 1, 1, 1, 3])
```

sage.graphs.distances_all_pairs.**shortest_path_all_pairs**(*G*)

Returns 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.

The integer corresponding to a vertex is its index in the list `G.vertices()`.

EXAMPLE:

```
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)

Returns the Wiener index of the graph.

The Wiener index of a graph G can be defined in two equivalent ways [KRG96b] :

- $W(G) = \frac{1}{2}\sum_{u,v\in G} d(u,v)$ 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 $W(G) = \sum_{e\in E(G)} |\Omega(e)|$.

EXAMPLE:

From [GYLL93c], cited in [KRG96b]:

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

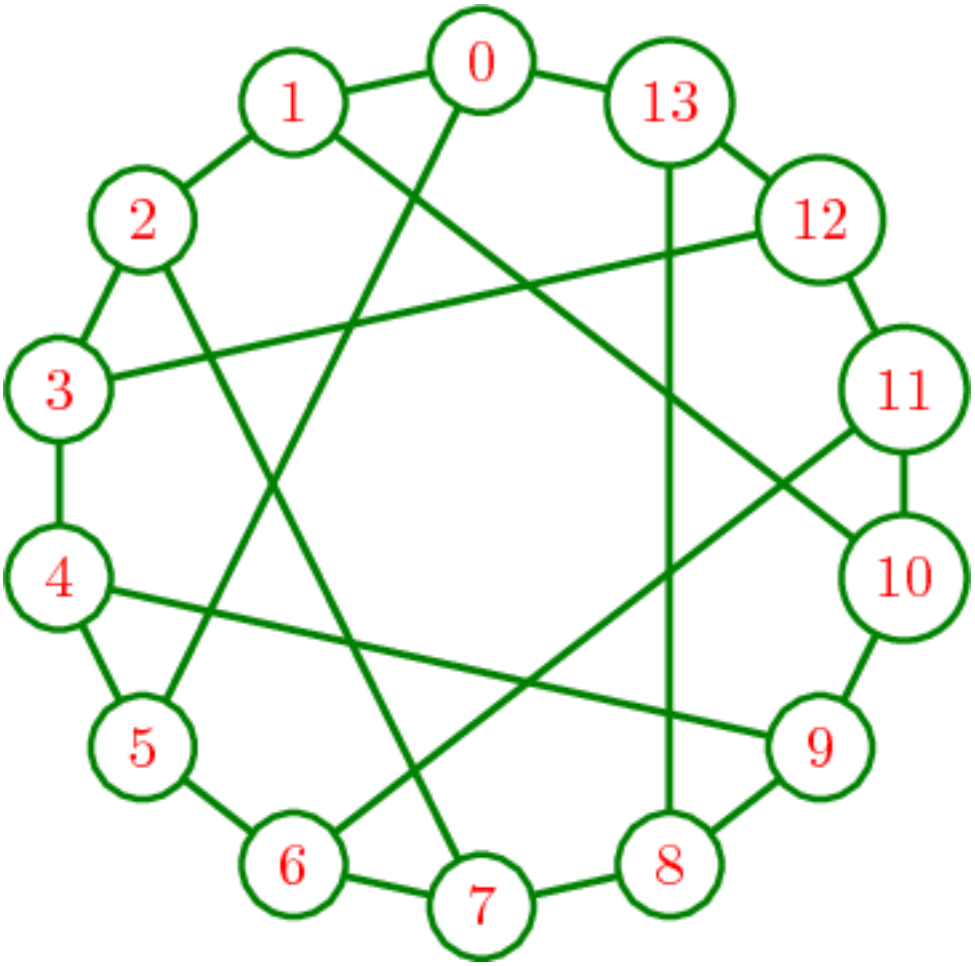
5.22 LaTeX options for graphs

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. Thiery (2010-02): dot2tex/graphviz interface
- Rob Beezer (2010-05-29): Extended range of `tkz-graph` options

5.22.1 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/graph.html

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}
%
\useasboundingbox (0,0) rectangle (5.0cm,5.0cm);
%
\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}
\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}
%
\Vertex[style={minimum size=0.2cm,draw=cv0,fill=cfv0,text=clv0,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv2,fill=cfv2,text=clv2,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv3,fill=cfv3,text=clv3,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv4,fill=cfv4,text=clv4,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv5,fill=cfv5,text=clv5,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv6,fill=cfv6,text=clv6,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv7,fill=cfv7,text=clv7,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv8,fill=cfv8,text=clv8,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv9,fill=cfv9,text=clv9,shape=circle},LabelOut=false,L=\hbox{$
\Vertex[style={minimum size=0.2cm,draw=cv10,fill=cfv10,text=clv10,shape=circle},LabelOut=false,L=\hbo
\Vertex[style={minimum size=0.2cm,draw=cv11,fill=cfv11,text=clv11,shape=circle},LabelOut=false,L=\hbo
\Vertex[style={minimum size=0.2cm,draw=cv12,fill=cfv12,text=clv12,shape=circle},LabelOut=false,L=\hbo
\Vertex[style={minimum size=0.2cm,draw=cv13,fill=cfv13,text=clv13,shape=circle},LabelOut=false,L=\hbo
%
\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') # optional - dot2tex
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: latex(g) # optional - dot2tex
\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=
sage: latex(g) # optional - dot2tex
\begin{tikzpicture}[>=latex,line join=bevel,]
...
\end{tikzpicture}
```

TEST:

This graph will look horrible, but it illustrates (and tests) a great variety of the possible options available through Sage's interface to the `tkz-graph` package. So it is worth viewing this in the notebook to see the effects of various defaults and choices.

```
sage: var('x y u w')
(x, y, u, w)
sage: G = Graph(loops=True)
sage: for i in range(5):
...       for j in range(i+1, 5):
...           G.add_edge((i, j), label=(x^i*y^j).expand())
sage: G.add_edge((0,0), label=sin(u))
sage: G.add_edge((4,4), label=w^5)
sage: G.set_pos(G.layout_circular())
sage: G.set_latex_options(
...   units='in',
...   graphic_size=(8,8),
...   margins=(1,2,2,1),
...   scale=0.5,
...   vertex_color='0.8',
...   vertex_colors={1:'aqua', 3:'y', 4:'#0000FF'},
...   vertex_fill_color='blue',
```

```
...    vertex_fill_colors={1:'green', 3:'b', 4:'#FF00FF'},
...    vertex_label_color='brown',
...    vertex_label_colors={0:'g',1:'purple',2:'#007F00'},
...    vertex_shape='diamond',
...    vertex_shapes={1:'rectangle', 2:'sphere', 3:'sphere', 4:'circle'},
...    vertex_size=0.3,
...    vertex_sizes={0:1.0, 2:0.3, 4:1.0},
...    vertex_label_placements = {2:(0.6, 180), 4:(0,45)},
...    edge_color='purple',
...    edge_colors={(0,2):'g',(3,4):'red'},
...    edge_fills=True,
...    edge_fill_color='green',
...    edge_label_colors={(2,3):'y',(0,4):'blue'},
...    edge_thickness=0.05,
...    edge_thicknesses={(3,4):0.2, (0,4):0.02},
...    edge_labels=True,
...    edge_label_sloped=True,
...    edge_label_slopes={(0,3):False, (2,4):False},
...    edge_label_placement=0.50,
...    edge_label_placements={(0,4):'above', (2,3):'left', (0,0):'above', (4,4):'below'},
...    loop_placement=(2.0, 'NO'),
...    loop_placements={4:(8.0, 'EA')}
...    )
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: print latex(G)
\begin{tikzpicture}
%
\useasboundingbox (0,0) rectangle (4.0in,4.0in);
%
\definecolor{cv0}{rgb}{0.8,0.8,0.8}
\definecolor{cfv0}{rgb}{0.0,0.0,1.0}
\definecolor{clv0}{rgb}{0.0,0.5,0.0}
\definecolor{cv1}{rgb}{0.0,1.0,1.0}
\definecolor{cfv1}{rgb}{0.0,0.502,0.0}
\definecolor{clv1}{rgb}{0.502,0.0,0.502}
\definecolor{cv2}{rgb}{0.8,0.8,0.8}
\definecolor{cfv2}{rgb}{0.0,0.0,1.0}
\definecolor{clv2}{rgb}{0.0,0.498,0.0}
\definecolor{cv3}{rgb}{0.75,0.75,0.0}
\definecolor{cfv3}{rgb}{0.0,0.0,1.0}
\definecolor{clv3}{rgb}{0.6471,0.1647,0.1647}
\definecolor{cv4}{rgb}{0.0,0.0,1.0}
\definecolor{cfv4}{rgb}{1.0,0.0,1.0}
\definecolor{clv4}{rgb}{0.6471,0.1647,0.1647}
\definecolor{cv0v0}{rgb}{0.502,0.0,0.502}
\definecolor{cfv0v0}{rgb}{0.0,0.502,0.0}
\definecolor{clv0v0}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v1}{rgb}{0.502,0.0,0.502}
\definecolor{cfv0v1}{rgb}{0.0,0.502,0.0}
\definecolor{clv0v1}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v2}{rgb}{0.0,0.5,0.0}
\definecolor{cfv0v2}{rgb}{0.0,0.502,0.0}
\definecolor{clv0v2}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v3}{rgb}{0.502,0.0,0.502}
\definecolor{cfv0v3}{rgb}{0.0,0.502,0.0}
\definecolor{clv0v3}{rgb}{0.0,0.0,0.0}
\definecolor{cv0v4}{rgb}{0.502,0.0,0.502}
```

```
\definecolor{cfv0v4}{rgb}{0.0,0.502,0.0}
\definecolor{clv0v4}{rgb}{0.0,0.0,1.0}
\definecolor{cv1v2}{rgb}{0.502,0.0,0.502}
\definecolor{cfv1v2}{rgb}{0.0,0.502,0.0}
\definecolor{clv1v2}{rgb}{0.0,0.0,0.0}
\definecolor{cv1v3}{rgb}{0.502,0.0,0.502}
\definecolor{cfv1v3}{rgb}{0.0,0.502,0.0}
\definecolor{clv1v3}{rgb}{0.0,0.0,0.0}
\definecolor{cv1v4}{rgb}{0.502,0.0,0.502}
\definecolor{cfv1v4}{rgb}{0.0,0.502,0.0}
\definecolor{clv1v4}{rgb}{0.0,0.0,0.0}
\definecolor{cv2v3}{rgb}{0.502,0.0,0.502}
\definecolor{cfv2v3}{rgb}{0.0,0.502,0.0}
\definecolor{clv2v3}{rgb}{0.75,0.75,0.0}
\definecolor{cv2v4}{rgb}{0.502,0.0,0.502}
\definecolor{cfv2v4}{rgb}{0.0,0.502,0.0}
\definecolor{clv2v4}{rgb}{0.0,0.0,0.0}
\definecolor{cv3v4}{rgb}{1.0,0.0,0.0}
\definecolor{cfv3v4}{rgb}{0.0,0.502,0.0}
\definecolor{clv3v4}{rgb}{0.0,0.0,0.0}
\definecolor{cv4v4}{rgb}{0.502,0.0,0.502}
\definecolor{cfv4v4}{rgb}{0.0,0.502,0.0}
\definecolor{clv4v4}{rgb}{0.0,0.0,0.0}
%
\Vertex[style={minimum size=0.5in,draw=cv0,fill=cfv0,text=clv0,shape=diamond},LabelOut=false,L=\hbox
\Vertex[style={minimum size=0.15in,draw=cv1,fill=cfv1,text=clv1,shape=rectangle},LabelOut=false,L=\hb
\Vertex[style={minimum size=0.15in,draw=cv2,fill=cfv2,text=clv2,shape=circle,shading=ball,line width=
\Vertex[style={minimum size=0.15in,draw=cv3,fill=cfv3,text=clv3,shape=circle,shading=ball,line width=
\Vertex[style={minimum size=0.5in,draw=cv4,fill=cfv4,text=clv4,shape=circle},LabelOut=true,Ldist=0.0
%
\Loop[dist=1.0in,dir=NO,style={color=cv0v0,double=cfv0v0},labelstyle={sloped,above,text=clv0v0,},labe
\Edge[lw=0.025in,style={color=cv0v1,double=cfv0v1},labelstyle={sloped,pos=0.5,text=clv0v1,},label=\hb
\Edge[lw=0.025in,style={color=cv0v2,double=cfv0v2},labelstyle={sloped,pos=0.5,text=clv0v2,},label=\hb
\Edge[lw=0.025in,style={color=cv0v3,double=cfv0v3},labelstyle={pos=0.5,text=clv0v3,},label=\hbox{$y^
\Edge[lw=0.01in,style={color=cv0v4,double=cfv0v4},labelstyle={sloped,above,text=clv0v4,},label=\hbox
\Edge[lw=0.025in,style={color=cv1v2,double=cfv1v2},labelstyle={sloped,pos=0.5,text=clv1v2,},label=\hb
\Edge[lw=0.025in,style={color=cv1v3,double=cfv1v3},labelstyle={sloped,pos=0.5,text=clv1v3,},label=\hb
\Edge[lw=0.025in,style={color=cv1v4,double=cfv1v4},labelstyle={sloped,pos=0.5,text=clv1v4,},label=\hb
\Edge[lw=0.025in,style={color=cv2v3,double=cfv2v3},labelstyle={sloped,left,text=clv2v3,},label=\hbox
\Edge[lw=0.025in,style={color=cv2v4,double=cfv2v4},labelstyle={pos=0.5,text=clv2v4,},label=\hbox{$x^
\Edge[lw=0.1in,style={color=cv3v4,double=cfv3v4},labelstyle={sloped,pos=0.5,text=clv3v4,},label=\hbox
\Loop[dist=4.0in,dir=EA,style={color=cv4v4,double=cfv4v4},labelstyle={sloped,below,text=clv4v4,},labe
%
\end{tikzpicture}
```

5.22.2 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()

Calls 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_3) at (...bp,...bp) [draw,draw=none] {$\left(1, 1\right)$};
  \node (node_2) at (...bp,...bp) [draw,draw=none] {$\left(1, 0\right)$};
  \node (node_1) at (...bp,...bp) [draw,draw=none] {$\left(0, 1\right)$};
  \node (node_0) at (...bp,...bp) [draw,draw=none] {$\left(0, 0\right)$};
  \draw [black,->] (node_0) ..controls (...bp,...bp) and (...bp,...bp)  .. (node_3);
  \draw [black,->] (node_2) ..controls (...bp,...bp) and (...bp,...bp)  .. (node_1);
  \draw [black,->] (node_0) ..controls (...bp,...bp) and (...bp,...bp)  .. (node_1);
  \draw [black,->] (node_2) ..controls (...bp,...bp) and (...bp,...bp)  .. (node_3);
%
\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_1) at (...bp,...bp) [draw,draw=none] {$3333$};
  \node (node_0) at (...bp,...bp) [draw,draw=none] {$88$};
  \draw [black,->] (node_1) ..controls (...bp,...bp) and (...bp,...bp)  .. (node_0);
  \definecolor{strokecol}{rgb}{0.0,0.0,0.0};
  \pgfsetstrokecolor{strokecol}
  \draw (...bp,...bp) node {$\text{\texttt{my{\char`\_}label}}$};
%
\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*)

Returns 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()

Returns 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()
sage: print opts.latex()
\begin{tikzpicture}
%
\useasboundingbox (0,0) rectangle (5.0cm,5.0cm);
%
\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,
\Vertex[style={minimum size=1.0cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},LabelOut=false,
%
\Edge[lw=0.1cm,style={color=cv0v1,},](v0)(v1)
%
\end{tikzpicture}
```

set_option(*option_name*, *option_value=None*)

Sets, modifies, clears 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. How-

ever, 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.

INPUTS:

- 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 – 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 – 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 – default: 'cm' – a natural unit of measurement used for all dimensions. Possible values are: 'in','mm','cm','pt', 'em', 'ex'
- scale – 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 – default: (5,5) – overall dimensions (width, length) of the bounding box around the entire graphic image
- margins – 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 tkzpicture.
- 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 – default: 'circle' – a string for 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– 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 verrices will not be either.
- vertex_sizes – a dictionary of sizes for some of the vertices.
- vertex_labels – default: True – a boolean to determine whether or not to display the vertex labels. If False subsequent options about vertex labels are ignored.
- vertex_labels_math – 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 spirt 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 – default: False – a boolean that determines if 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 theis 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 – default: 0.1 - a number specifying 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 – default: False – a boolean that determines if edge labels are shown. If False subsequent options about edge labels are ignored.
- edge_labels_math – default: True – a boolean that controls 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 – default: True a boolean that 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 – 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') – a pair, that determines 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 – 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 – a 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 - a boolean (default: False). Colors the edges according to their labels

OUTPUTS:

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. Further more, 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

TESTS:

These test all of the options and one example of each allowable proper input. They should all execute silently.

```
sage: G=Graph()
sage: G.add_edge((0,1))
sage: opts = G.latex_options()
sage: opts.set_option('tkz_style', 'Custom')
sage: opts.set_option('tkz_style', 'Art')
sage: opts.set_option('format', 'tkz_graph')
sage: opts.set_option('layout', 'acyclic')
sage: opts.set_option('prog', 'dot')
sage: opts.set_option('units', 'cm')
sage: opts.set_option('scale', 1.0)
sage: opts.set_option('graphic_size', (5, 5))
sage: opts.set_option('margins', (0,0,0,0))
sage: opts.set_option('vertex_color', 'black')
sage: opts.set_option('vertex_colors', {0:'#ABCDEF'})
```

```
sage: opts.set_option('vertex_fill_color', 'white')
sage: opts.set_option('vertex_fill_colors', {0:'c'})
sage: opts.set_option('vertex_shape', 'circle')
sage: opts.set_option('vertex_shapes', {0:'sphere'})
sage: opts.set_option('vertex_size', 1.0)
sage: opts.set_option('vertex_sizes', {0:3.4})
sage: opts.set_option('vertex_labels', True)
sage: opts.set_option('vertex_labels_math', True)
sage: opts.set_option('vertex_label_color', 'black')
sage: opts.set_option('vertex_label_colors', {0:'.23'})
sage: opts.set_option('vertex_label_placement', 'center')
sage: opts.set_option('vertex_label_placement', (3, 4.2))
sage: opts.set_option('vertex_label_placements', {0:'center'})
sage: opts.set_option('vertex_label_placements', {0:(4.7,1)})
sage: opts.set_option('edge_color', 'black')
sage: opts.set_option('edge_colors', {(0,1):'w'})
sage: opts.set_option('edge_fills', False)
sage: opts.set_option('edge_fill_color', 'black')
sage: opts.set_option('edge_fill_colors', {(0,1):"#123456"})
sage: opts.set_option('edge_thickness', 0.1)
sage: opts.set_option('edge_thicknesses', {(0,1):5.2})
sage: opts.set_option('edge_labels', False)
sage: opts.set_option('edge_labels_math', True)
sage: opts.set_option('edge_label_color', 'black')
sage: opts.set_option('edge_label_colors', {(0,1):'red'})
sage: opts.set_option('edge_label_sloped', True)
sage: opts.set_option('edge_label_slopes', {(0,1): False})
sage: opts.set_option('edge_label_placement', 'left')
sage: opts.set_option('edge_label_placement', 0.50)
sage: opts.set_option('edge_label_placements', {(0,1):'above'})
sage: opts.set_option('edge_label_placements', {(0,1):0.75})
sage: opts.set_option('loop_placement', (3.0, 'NO'))
sage: opts.set_option('loop_placements', {0:(5.7,'WE')})
```

These test some of the logic of possible failures. Some tests, such as inputs of colors, are handled by somewhat general sections of code and are not tested for each possible option.

```
sage: G=Graph()
sage: G.add_edge((0,1))
sage: opts = G.latex_options()
sage: opts.set_option('tkz_style', 'Crazed')
Traceback (most recent call last):
...
ValueError: tkz_style is not "Custom", nor an implemented tkz-graph style
sage: opts.set_option('format', 'NonExistent')
Traceback (most recent call last):
...
ValueError: format option must be one of: tkz_graph, dot2tex not NonExistent
sage: opts.set_option('units', 'furlongs')
Traceback (most recent call last):
...
ValueError: units option must be one of: in, mm, cm, pt, em, ex, not furlongs
sage: opts.set_option('graphic_size', (1,2,3))
Traceback (most recent call last):
...
ValueError: graphic_size option must be an ordered pair, not (1, 2, 3)
sage: opts.set_option('margins', (1,2,3))
Traceback (most recent call last):
```

```
...
ValueError: margins option must be 4-tuple, not (1, 2, 3)
sage: opts.set_option('vertex_color', 'chartruse')
Traceback (most recent call last):
...
ValueError: vertex_color option needs to be a matplotlib color (always as a string), not cha
sage: opts.set_option('vertex_labels_math', 'maybe')
Traceback (most recent call last):
...
ValueError: vertex_labels_math option must be True or False, not maybe
sage: opts.set_option('vertex_shape', 'decagon')
Traceback (most recent call last):
...
ValueError: vertex_shape option must be the shape of a vertex, not decagon
sage: opts.set_option('scale', 'big')
Traceback (most recent call last):
...
ValueError: scale option must be a positive number, not big
sage: opts.set_option('scale', -6)
Traceback (most recent call last):
...
ValueError: scale option must be a positive number, not -6
sage: opts.set_option('vertex_label_placement', (2,-4))
Traceback (most recent call last):
...
ValueError: vertex_label_placement option must be None, or a pair of positive numbers, not (
sage: opts.set_option('edge_label_placement', 3.6)
Traceback (most recent call last):
...
ValueError: edge_label_placement option must be a number between 0.0 and 1.0 or a place (lik
sage: opts.set_option('loop_placement', (5,'SW'))
Traceback (most recent call last):
...
ValueError: loop_placement option must be a pair that is a positive number followed by a com
sage: opts.set_option('vertex_fill_colors', {0:'#GG0000'})
Traceback (most recent call last):
...
ValueError: vertex_fill_colors option for 0 needs to be a matplotlib color (always as a stri
sage: opts.set_option('vertex_sizes', {0:-10})
Traceback (most recent call last):
...
ValueError: vertex_sizes option for 0 needs to be a positive number, not -10
sage: opts.set_option('edge_label_slopes', {(0,1):'possibly'})
Traceback (most recent call last):
...
ValueError: edge_label_slopes option for (0, 1) needs to be True or False, not possibly
sage: opts.set_option('vertex_shapes', {0:'pentagon'})
Traceback (most recent call last):
...
ValueError: vertex_shapes option for 0 needs to be a vertex shape, not pentagon
sage: opts.set_option('vertex_label_placements', {0:(1,2,3)})
Traceback (most recent call last):
...
ValueError: vertex_label_placements option for 0 needs to be None or a pair of positive numb
sage: opts.set_option('edge_label_placements', {(0,1):'partway'})
Traceback (most recent call last):
...
ValueError: edge_label_placements option for (0, 1) needs to be a number between 0.0 and 1.0
sage: opts.set_option('loop_placements', {0:(-3,'WE')})
```

```
Traceback (most recent call last):
...
ValueError: loop_placements option for 0 needs to be a positive number and a compass point (
sage: opts.set_option('margins', (1,2,3,-5))
Traceback (most recent call last):
...
ValueError: margins option of (1, 2, 3, -5) cannot contain -5
```

set_options(***kwds*)

Set several LaTeX options for a graph all at once.

INPUTS:

- kwds - any number of option/value pairs to se 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]
%
\useasboundingbox (0,0) rectangle (5.0cm,5.0cm);
%
\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}
%
\useasboundingbox (0,0) rectangle (5.0cm,5.0cm);
%
\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,
\Vertex[style={minimum size=1.0cm,draw=cv1,fill=cfv1,text=clv1,shape=circle},LabelOut=false,
\Vertex[style={minimum size=1.0cm,draw=cv2,fill=cfv2,text=clv2,shape=circle},LabelOut=false,
%
\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.

TESTS:

Graphs with preset layouts that are vertical or horizontal can cause problems. First test is a horizontal layout on a path with three vertices.

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: g = graphs.PathGraph(3)
sage: opts = g.latex_options()
sage: print opts.tkz_picture()
\begin{tikzpicture}
...
\end{tikzpicture}
```

Scaling to a bounding box is problematic for graphs with just one vertex, or none.

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: g = graphs.CompleteGraph(1)
sage: opts = g.latex_options()
sage: print opts.tkz_picture()
\begin{tikzpicture}
...
\end{tikzpicture}
```

sage.graphs.graph_latex.**check_tkz_graph**()

Checks 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()`

TESTS:

```
sage: from sage.graphs.graph_latex import check_tkz_graph
sage: check_tkz_graph()  # random - depends on TeX installation
sage: check_tkz_graph()  # at least the second time, so no output
```

sage.graphs.graph_latex.**have_tkz_graph**()

Returns `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()`

TESTS:

```
sage: from sage.graphs.graph_latex import have_tkz_graph
sage: have_tkz_graph()  # random - depends on TeX installation
sage: have_tkz_graph() in [True, False]
True
```

sage.graphs.graph_latex.**setup_latex_preamble**()

Adds 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()
```

TESTS:

```
sage: ("\usepackage{tikz}" in latex.extra_preamble()) == latex.has_file("tikz.sty")
True
```

5.23 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],[0.0,0.49999999999999
sage: graph_to_js(graphs.StarGraph(2))
'num_vertices=3;edges=[[0,1],[0,2]];pos=[[0.75,0.5],[1.0,0.0],[0.0,1.0]];'
```

5.24 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*)

Returns 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.

EXAMPLE:

```
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*)
Returns 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.

EXAMPLE:

```
sage: l = [':P_`cBaC_ACd`C_@BC`ABDHaEH_@BF_@CHIK_@BCEHKL_BIKM_BFGHI',':f`??KO?B_OOSCGE_?OWONDBO?
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*)
Returns a list of Sage Graphs, given a list of whatever kind of data.

INPUT:

- `data` - can be a string, a list of strings, or a file stream, or whatever.

EXAMPLE:

```
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.graph_list.**show_graphs**(*list*, ***kwds*)
Shows a maximum of 20 graphs from 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 if 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:

- `list` - a 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)
```

Here's an 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_arrays

```
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**(*list*, *file=None*, *output_list=False*)

Converts 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:

- `list` - a Python list of Sage Graphs
- `file` - (optional) a file stream to write to (must be in 'w' mode)
- `output_list` - False - output is a string True - output is a list of strings (ignored if file gets specified)

EXAMPLE:

```
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_arrays**(*list*, ***kwds*)

Returns a list of Sage graphics arrays containing the graphs in list. The maximum number of graphs per array is 20 (5 rows of 4). Use this function if there are too many graphs for the show_graphs function. The graphics arrays will contain 20 graphs each except potentially the last graphics array in the list.

INPUT:

- `list` - a 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_arrays(glist)
```

Display the first graphics array in the list.

```
sage: garray[0].show()
```

Display the last graphics array in the list.

```
sage: garray[len(garray)-1].show()
```

See the .plot() or .show() documentation for an individual graph for options, all of which are available from to_graphics_arrays

```
sage: glist = []
sage: for _ in range(10):
...       glist.append(graphs.RandomLobster(41, .3, .4))
sage: w = graphs_list.to_graphics_arrays(glist, layout='spring', vertex_size=20)
sage: len(w)
1
sage: w[0]
```

sage.graphs.graph_list.**to_sparse6**(*list*, *file=None*, *output_list=False*)

Converts 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:

- `list` - a Python list of Sage Graphs
- `file` - (optional) a file stream to write to (must be in 'w' mode)
- `output_list` - False - output is a string True - output is a list of strings (ignored if file gets specified)

EXAMPLE:

```
sage: l = [graphs.DodecahedralGraph(), graphs.PetersenGraph()]
sage: graphs_list.to_sparse6(l)
':S_`abcaDe`Fg_HijhKfLdMkNcOjP_BQ\n:I`ES@obGkqegW~\n'
```

5.25 Hyperbolicity

Hyperbolicity

Definition :

The hyperbolicity δ of a graph G has been defined by Gromov [Gromov87] 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

$$S_1 = dist(a, b) + dist(b, c)$$
$$S_2 = dist(a, c) + dist(b, d)$$
$$S_3 = dist(a, d) + dist(b, c)$$

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 [FIV12]. This remains very long for large-scale graphs, and much harder to implement.

An improvement over the naive algorithm has been proposed in [CCL12], and is implemented in the current module. Like the naive algorithm, it has complexity $O(n^4)$ but behaves much better in practice. It uses the following fact :

Assume that $S_1 = dist(a,b) + dist(c,d)$ is the largest 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, we ensure that S_1 is larger than S_2 and S_3 using an ordering of the pairs by decreasing lengths. Furthermore, we use the best value h found so far to cut exploration.

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.

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.

REFERENCES:

AUTHORS:

- David Coudert (2012): initial version, exact and approximate algorithm, distribution, sampling

5.25.1 Methods

sage.graphs.hyperbolicity.**elimination_ordering_of_simplicial_vertices**(*G*, *max_degree=4*, *verbose=False*)

Return an elimination ordering of simplicial vertices.

An elimination ordering of simplicial vertices is an elimination ordering of the vertices of the graphs such that the induced subgraph of their neighbors is a clique. More precisely, as long as the graph has a vertex `u` such that the induced subgraph of its neighbors is a clique, we remove `u` from the graph, add it to the elimination ordering (list of vertices), and repeat. This method is inspired from the decomposition of a graph by clique-separators.

INPUTS:

- `G` – a Graph
- `max_degree` – (default: 4) maximum degree of the vertices to consider. The running time of this method depends on the value of this parameter.
- `verbose` – (default: `False`) is boolean set to `True` to display some information during execution.

OUTPUT:

- `elim` – A ordered list of vertices such that vertex `elim[i]` is removed before vertex `elim[i+1]`.

TESTS:

Giving anything else than a Graph:

```
sage: from sage.graphs.hyperbolicity import elimination_ordering_of_simplicial_vertices
sage: elimination_ordering_of_simplicial_vertices([])
Traceback (most recent call last):
...
ValueError: The input parameter must be a Graph.
```

Giving two small bounds on degree:

```
sage: from sage.graphs.hyperbolicity import elimination_ordering_of_simplicial_vertices
sage: elimination_ordering_of_simplicial_vertices(Graph(), max_degree=0)
Traceback (most recent call last):
...
ValueError: The parameter max_degree must be > 0.
```

Giving a graph built from a bipartite graph plus an edge:

```
sage: G = graphs.CompleteBipartiteGraph(2,10)
sage: G.add_edge(0,1)
sage: from sage.graphs.hyperbolicity import elimination_ordering_of_simplicial_vertices
sage: elimination_ordering_of_simplicial_vertices(G)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 1, 11]
sage: elimination_ordering_of_simplicial_vertices(G,max_degree=1)
[]
```

sage.graphs.hyperbolicity.**hyperbolicity**(*G*, *algorithm='cuts'*, *approximation_factor=None*, *additive_gap=None*, *verbose=False*)

Return the hyperbolicity of the graph or an approximation of this value.

The hyperbolicity of a graph has been defined by Gromov [Gromov87] 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 Graph
- `algorithm` – (default: `'cuts'`) specifies the algorithm to use among:
  - `'basic'` is an exhaustive algorithm considering all possible 4-tuples and so have time complexity in $O(n^4)$.
  - `'cuts'` is an exact algorithm proposed in [CCL12]. 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.
  - `'cuts+'` is an additive constant approximation algorithm. It proceeds by first removing the simplicial vertices and then applying the `'cuts'` algorithm on the remaining graph, as documented in [CCL12]. By default, the additive constant of the approximation is one. This value can be increased by setting the `additive_gap` to the desired value, provide `additive_gap \geq 1`. In some cases, the returned result is proven optimal. However, this algorithm *cannot* be used to compute an approximation with multiplicative factor, and so the `approximation_factor` parameter is just ignored here.
  - `'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 `'cuts'` 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 optimaly. This parameter is used only when the chosen algorithm is `'cuts'`.
- `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 optimaly. This parameter is used only when the chosen algorithm is `'cuts'` or `'cuts+'`. The parameter must be `\geq 1` when used with `'cuts+'`.
- `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.GridGraph([3,3])
sage: hyperbolicity(G,algorithm='cuts')
(2, [(0, 0), (0, 2), (2, 0), (2, 2)], 2)
```

```
sage: hyperbolicity(G,algorithm='basic')
(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: hyperbolicity(G,algorithm='cuts')
(1/2, [0, 1, 2, 3], 1/2)
sage: hyperbolicity(G,algorithm='basic')
(1/2, [0, 1, 2, 3], 1/2)
sage: hyperbolicity(G,algorithm='dom')
(0, [0, 2, 8, 9], 1)
```

Asking for an approximation:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.GridGraph([2,10])
sage: hyperbolicity(G,algorithm='cuts', approximation_factor=1.5)
(1, [(0, 0), (0, 9), (1, 0), (1, 9)], 3/2)
sage: hyperbolicity(G,algorithm='cuts', approximation_factor=4)
(1, [(0, 0), (0, 9), (1, 0), (1, 9)], 4)
sage: hyperbolicity(G,algorithm='cuts', additive_gap=2)
(1, [(0, 0), (0, 9), (1, 0), (1, 9)], 3)
sage: hyperbolicity(G,algorithm='cuts+')
(1, [(0, 0), (0, 9), (1, 0), (1, 9)], 2)
sage: hyperbolicity(G,algorithm='cuts+', additive_gap=2)
(1, [(0, 0), (0, 9), (1, 0), (1, 9)], 3)
sage: hyperbolicity(G,algorithm='dom')
(1, [(0, 1), (0, 9), (1, 0), (1, 8)], 5)
```

Comparison of results:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: for i in xrange(10): # long time
...       G = graphs.RandomBarabasiAlbert(100,2)
...       d1,_,_ = hyperbolicity(G,algorithm='basic')
...       d2,_,_ = hyperbolicity(G,algorithm='cuts')
...       d3,_,_ = hyperbolicity(G,algorithm='cuts+')
...       l3,_,u3 = hyperbolicity(G,approximation_factor=2)
...       if d1!=d2 or d1<d3 or l3>d1 or u3<d1:
...          print "That's not good!"
```

TESTS:

Giving anything else than a Graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: hyperbolicity([])
Traceback (most recent call last):
...
ValueError: The input parameter must be a Graph.
```

Giving a non connected graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = Graph([(0,1),(2,3)])
sage: hyperbolicity(G)
Traceback (most recent call last):
...
```

```
ValueError: The input Graph must be connected.
```

Giving wrong approximation factor:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = graphs.PetersenGraph()
sage: hyperbolicity(G,algorithm='cuts', approximation_factor=0.1)
Traceback (most recent call last):
...
ValueError: The approximation factor must be >= 1.0.
```

Giving negative additive gap:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = Graph()
sage: hyperbolicity(G,algorithm='cuts', additive_gap=-1)
Traceback (most recent call last):
...
ValueError: The additive gap must be >= 0 when using the 'cuts' algorithm.
```

Asking for an unknown algorithm:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity
sage: G = Graph()
sage: hyperbolicity(G,algorithm='tip top')
Traceback (most recent call last):
...
ValueError: Algorithm 'tip top' not yet implemented. Please contribute.
```

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 [Gromov87] 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 dictionnary 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}
```

TESTS:

Giving anythin else than a Graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity_distribution
sage: hyperbolicity_distribution([])
Traceback (most recent call last):
...
ValueError: The input parameter must be a Graph.
```

Giving a non connected graph:

```
sage: from sage.graphs.hyperbolicity import hyperbolicity_distribution
sage: G = Graph([(0,1),(2,3)])
sage: hyperbolicity_distribution(G)
Traceback (most recent call last):
...
ValueError: The input Graph must be connected.
```

5.26 Tutte polynomial

This module implements a deletion-contraction algorithm for computing the Tutte polynomial as described in the paper [Gordon10].

`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

5.26.1 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.

REFERENCES:

5.26.2 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 [Gordon10]:

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:

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.edges
[(0, 1), (1, 2), (2, 3)]
```

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.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
```

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

x.__init__(...) initializes x; see help(type(x)) for signature

class sage.graphs.tutte_polynomial.**MaximizeDegree**
Bases: sage.graphs.tutte_polynomial.EdgeSelection

x.__init__(...) initializes x; see help(type(x)) for signature

class sage.graphs.tutte_polynomial.**MinimizeDegree**
Bases: sage.graphs.tutte_polynomial.EdgeSelection

x.__init__(...) initializes x; see help(type(x)) for signature

class sage.graphs.tutte_polynomial.**MinimizeSingleDegree**
Bases: sage.graphs.tutte_polynomial.EdgeSelection

x.__init__(...) initializes x; see help(type(x)) for signature

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*)
Returns the a 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).iteritems())
[((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 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]
[(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),2) as Y:
....:     G.edges()
[]
sage: G.edges()
[(0, 1, None), (0, 1, None)]
```

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()
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)
```

TESTS:

Providing an external cache:

```
sage: cache = {}
sage: _ = graphs.RandomGNP(7,.5).tutte_polynomial(cache=cache)
sage: len(cache) > 0
True
```

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)]
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[GYLL93] I. Gutman, Y.-N. Yeh, S.-L. Lee, and Y.-L. Luo. Some recent results in the theory of the Wiener number. *Indian Journal of Chemistry*, 32A:651–661, 1993.

[Tarjan72] R.E. Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1(2): 146-160 (1972).

[HarPri] F. Harary and G. Prins. The block-cutpoint-tree of a graph. Publ. Math. Debrecen 13 1966 103-107.

[Gallai] T. Gallai, Elementare Relationen bezueglich der Glieder und trennenden Punkte von Graphen, Magyar Tud. Akad. Mat. Kutato Int. Kozl. 9 (1964) 235-236

[HSSNX] Aric Hagberg, Dan Schult and Pieter Swart. NetworkX documentation. [Online] Available: http://networkx.github.io/documentation/latest/reference/index.html

[LPForm] Nathann Cohen, Several Graph problems and their Linear Program formulations, http://hal.archives-ouvertes.fr/inria-00504914/en

[KaisPacking] Thomas Kaiser A short proof of the tree-packing theorem Arxiv 0911.2809

[SchrijverCombOpt] Alexander Schrijver Combinatorial optimization: polyhedra and efficiency 2003

[dotspec] http://www.graphviz.org/doc/info/lang.html

[Rose75] Rose, D.J. and Tarjan, R.E., Algorithmic aspects of vertex elimination, Proceedings of seventh annual ACM symposium on Theory of computing Page 254, ACM 1975

[Fulkerson65] Fulkerson, D.R. and Gross, OA Incidence matrices and interval graphs Pacific J. Math 1965 Vol. 15, number 3, pages 835–855

[BM04] John M. Boyer and Wendy J. Myrvold, On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. Journal of Graph Algorithms and Applications, Vol. 8, No. 3, pp. 241-273, 2004.

[Kirkman] Kirkman, Emily A. O(n) Circular Planarity Testing. [Online] Available: soon!

[erdos1978choos] Erdos, P. and Rubin, A.L. and Taylor, H. Proc. West Coast Conf. on Combinatorics Graph Theory and Computing, Congressus Numerantium vol 26, pages 125–157, 1979

[Brandes2003] Ulrik Brandes. (2003). Faster Evaluation of Shortest-Path Based Centrality Indices. [Online] Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.1504

[Borgatti95] Stephen P Borgatti. (1995). Centrality and AIDS. [Online] Available: http://www.analytictech.com/networks/centaids.htm

[BroKer1973] Coen Bron and Joep Kerbosch. (1973). Algorithm 457: Finding All Cliques of an Undirected Graph. Commun. ACM. v 16. n 9. pages 575-577. ACM Press. [Online] Available: http://www.ram.org/computing/rambin/rambin.html

[WPkcore] K-core. Wikipedia. (2007). [Online] Available: Wikipedia article K-core

[PSW1996] Boris Pittel, Joel Spencer and Nicholas Wormald. Sudden Emergence of a Giant k-Core in a Random Graph. (1996). J. Combinatorial Theory. Ser B 67. pages 111-151. [Online] Available: http://cs.nyu.edu/cs/faculty/spencer/papers/k-core.pdf

[BZ] Vladimir Batagelj and Matjaz Zaversnik. An $O(m)$ Algorithm for Cores Decomposition of Networks. Arxiv cs/0310049v1.

[HST] Matthew D. Horton, H. M. Stark, and Audrey A. Terras, What are zeta functions of graphs and what are they good for? in Quantum graphs and their applications, 173-189, Contemp. Math., Vol. 415

[Terras] Audrey Terras, Zeta functions of graphs: a stroll through the garden, Cambridge Studies in Advanced Mathematics, Vol. 128

[ScottStorm] Geoffrey Scott and Christopher Storm, The coefficients of the Ihara zeta function, Involve (http://msp.org/involve/2008/1-2/involve-v1-n2-p08-p.pdf)

[AhaBerZiv07] R. Aharoni and E. Berger and R. Ziv Independent systems of representatives in weighted graphs Combinatorica vol 27, num 3, p253–267 2007

[ABCHRS08] L. Addario-Berry, M. Chudnovsky, F. Havet, B. Reed, P. Seymour Bisimplicial vertices in even-hole-free graphs Journal of Combinatorial Theory, Series B vol 98, n.6 pp 1119-1164, 2008

[CRST06] M. Chudnovsky, G. Cornuejols, X. Liu, P. Seymour, K. Vuskovic Recognizing berge graphs Combinatorica vol 25, n 2, pages 143–186 2005

[SPGT] M. Chudnovsky, N. Robertson, P. Seymour, R. Thomas. The strong perfect graph theorem Annals of Mathematics vol 164, number 1, pages 51–230 2006

[GraphClasses] A. Brandstadt, VB Le and JP Spinrad Graph classes: a survey SIAM Monographs on Discrete Mathematics and Applications}, 1999

[Marcolli2009] Matilde Marcolli, Feynman Motives, Chapter 3, Feynman integrals and algebraic varieties, http://www.its.caltech.edu/~matilde/LectureN3.pdf

[Brown2011] Francis Brown, Multiple zeta values and periods: From moduli spaces to Feynman integrals, in Contemporary Mathematics vol 539

[AMOZ06] Asahiro, Y. and Miyano, E. and Ono, H. and Zenmyo, K. Graph orientation algorithms to minimize the maximum outdegree Proceedings of the 12th Computing: The Australasian Theory Symposium Volume 51, page 20 Australian Computer Society, Inc. 2006

[FMDec] Fabien de Montgolfier http://www.liafa.jussieu.fr/~fm/algos/index.html

[HabibViennot1999] Michel Habib, Christiphe Paul, Laurent Viennot Partition refinement techniques: An interesting algorithmic tool kit International Journal of Foundations of Computer Science vol. 10 n2 pp.147–170, 1999

[CapHabMont02] C. Capelle, M. Habib et F. de Montgolfier Graph decomposition and Factorising Permutations Discrete Mathematics and Theoretical Computer Sciences, vol 5 no. 1 , 2002.

[HabPau10] Michel Habib and Christophe Paul A survey of the algorithmic aspects of modular decomposition Computer Science Review vol 4, number 1, pages 41–59, 2010 http://www.lirmm.fr/~paul/md-survey.pdf

[Har62] Harary, F (1962). The determinant of the adjacency matrix of a graph, SIAM Review 4, 202-210

[Biggs93] Biggs, N. L. Algebraic Graph Theory, 2nd ed. Cambridge, England: Cambridge University Press, pp. 45, 1993.

[RT75] Read, R. C. and Tarjan, R. E. Bounds on Backtrack Algoritms for Listing Cycles, Paths, and Spanning Trees Networks, Volume 5 (1975), numer 3, pages 237-252.

[ACFLSS04] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons: Kernelization Algorithm for the Vertex Cover Problem: Theory and Experiments. *SIAM ALENEX/ANALCO* 2004: 62-69.

[FAGDC] Fifth Annual Graph Drawing Contest P. Eaded, J. Marks, P.Mutzel, S. North http://www.merl.com/papers/docs/TR98-16.pdf

[CharLes1996] Chartrand, G. and Lesniak, L.: Graphs and Digraphs. Chapman and Hall/CRC, 1996.

[Newman2003] Newman, M.E.J. The Structure and function of complex networks, SIAM Review vol. 45, no. 2 (2003), pp. 167-256.

[ChungLu2002] Chung, Fan and Lu, L. Connected components in random graphs with given expected degree sequences. Ann. Combinatorics (6), 2002 pp. 125-145.

[ARETT-DOREE] Arett, Danielle and Doree, Suzanne "Coloring and counting on the Hanoi graphs" Mathematics Magazine, Volume 83, Number 3, June 2010, pages 200-9

[BROUWER-HS-2009] Higman-Sims graph. Andries E. Brouwer, accessed 24 October 2009.

[HIGMAN1968] A simple group of order 44,352,000, Math.Z. 105 (1968) 110-113. D.G. Higman & C. Sims.

[HAFNER2004] On the graphs of Hoffman-Singleton and Higman-Sims. The Electronic Journal of Combinatorics 11 (2004), #R77, Paul R. Hafner, accessed 24 October 2009.

[GodsilRoyle] Godsil, C. and Royle, G. Algebraic Graph Theory. Springer, 2001.

[ErdRen1959] P. Erdos and A. Renyi. On Random Graphs, Publ. Math. 6, 290 (1959).

[Gilbert1959] E. N. Gilbert. Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[BatBra2005] V. Batagelj and U. Brandes. Efficient generation of large random networks. Phys. Rev. E, 71, 036113, 2005.

[HolmeKim2002] Holme, P. and Kim, B.J. Growing scale-free networks with tunable clustering, Phys. Rev. E (2002). vol 65, no 2, 026107.

[boucheron2001] Boucheron, S. and FERNANDEZ de la VEGA, W., On the Independence Number of Random Interval Graphs, Combinatorics, Probability and Computing v10, issue 05, Pages 385–396, Cambridge Univ Press, 2001

[NWS99] Newman, M.E.J., Watts, D.J. and Strogatz, S.H. Random graph models of social networks. Proc. Nat. Acad. Sci. USA 99, 2566-2572.

[KimVu2003] Kim, Jeong Han and Vu, Van H. Generating random regular graphs. Proc. 35th ACM Symp. on Thy. of Comp. 2003, pp 213-222. ACM Press, San Diego, CA, USA. http://doi.acm.org/10.1145/780542.780576

[StegerWormald1999] Steger, A. and Wormald, N. Generating random regular graphs quickly. Prob. and Comp. 8 (1999), pp 377-396.

[CFHM12] On the Hyperbolicity of Small-World and Tree-Like Random Graphs Wei Chen, Wenjie Fang, Guangda Hu, Michael W. Mahoney http://arxiv.org/abs/1201.1717

[BCN89] A. E. Brouwer, A. M. Cohen, A. Neumaier, Distance-Regular Graphs, Springer, 1989.

[CIA] CIA Factbook 09 https://www.cia.gov/library/publications/the-world-factbook/

[buckygen] G. Brinkmann, J. Goedgebeur and B.D. McKay, Generation of Fullerenes, Journal of Chemical Information and Modeling, 52(11):2910-2918, 2012.

[benzene] G. Brinkmann, G. Caporossi and P. Hansen, A Constructive Enumeration of Fusenes and Benzenoids, Journal of Algorithms, 45:155-166, 2002.

[RPK80] S. M. Reddy, D. K. Pradhan, and J. Kuhl. Directed graphs with minimal diameter and maximal connectivity, School Eng., Oakland Univ., Rochester MI, Tech. Rep., July 1980.

[RPK83] S. Reddy, P. Raghavan, and J. Kuhl. A Class of Graphs for Processor Interconnection. *IEEE International Conference on Parallel Processing*, pages 154-157, Los Alamitos, Ca., USA, August 1983.

[II83] M. Imase and M. Itoh. A design for directed graphs with minimum diameter, *IEEE Trans. Comput.*, vol. C-32, pp. 782-784, 1983.

[Kautz68] W. H. Kautz. Bounds on directed (d, k) graphs. Theory of cellular logic networks and machines, AFCRL-68-0668, SRI Project 7258, Final Rep., pp. 20-28, 1968.

[Aki80] Akiyama, J. and Exoo, G. and Harary, F. Covering and packing in graphs. III: Cyclic and acyclic invariants Mathematical Institute of the Slovak Academy of Sciences Mathematica Slovaca vol30, n4, pages 405–417, 1980

[NisOst2003] Sampo Niskanen and Patric R. J. Ostergard, "Cliquer User's Guide, Version 1.0," Communications Laboratory, Helsinki University of Technology, Espoo, Finland, Tech. Rep. T48, 2003.

[ATGA] Advanced Topics in Graph Algorithms, Ron Shamir, http://www.cs.tau.ac.il/~rshamir/atga/atga.html

[Cleanup] A cleanup on transitive orientation, Orders, Algorithms, and Applications, 1994, Simon, K. and Trunz, P., ftp://ftp.inf.ethz.ch/doc/papers/ti/ga/ST94.ps.gz

[Whitney32] Congruent graphs and the connectivity of graphs, Whitney, American Journal of Mathematics, pages 150–168, 1932, available on JSTOR

[Harary69] Graph Theory, Harary, Addison-Wesley, 1969

[Beineke70] Lowell Beineke, Characterizations of derived graphs, Journal of Combinatorial Theory, Vol. 9(2), pages 129-135, 1970 http://dx.doi.org/10.1016/S0021-9800(70)80019-9

[CormenEtAl2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd edition, The MIT Press, 2001.

[GoodrichTamassia2001] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. 2nd edition, John Wiley & Sons, 2001.

[JoynerNguyenCohen2010] David Joyner, Minh Van Nguyen, and Nathann Cohen. *Algorithmic Graph Theory*. 2010, http://code.google.com/p/graph-theory-algorithms-book/

[Sahni2000] Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw-Hill, 2000.

[Haj] M. Hajiaghayi http://www-math.mit.edu/~hajiagha/pp11.ps

[WRIGHT-ETAL] Wright, Robert Alan; Richmond, Bruce; Odlyzko, Andrew; McKay, Brendan D. Constant time generation of free trees. SIAM J. Comput. 15 (1986), no. 2, 540–548.

[Godsil93] Chris Godsil (1993) Algebraic Combinatorics.

[Bod98] *A partial k-arboretum of graphs with bounded treewidth*, Hans L. Bodlaender, Theoretical Computer Science 209(1-2):1-45, 1998.

[Kin92] *The vertex separation number of a graph equals its path-width*, Nancy G. Kinnersley, Information Processing Letters 42(6):345-350, 1992.

[SP10] *Lightpath Reconfiguration in WDM networks*, Fernando Solano and Michal Pioro, IEEE/OSA Journal of Optical Communication and Networking 2(12):1010-1021, 2010.

[RWKlause] Philipp Klaus Krause – rw v0.2 http://pholia.tdi.informatik.uni-frankfurt.de/~philipp/software/rw.shtml

[Oum] Sang-il Oum Computing rank-width exactly Information Processing Letters, 2008 vol. 109, n. 13, p. 745–748 Elsevier http://mathsci.kaist.ac.kr/~sangil/pdf/2008exp.pdf

[BL] Buckles, B.P. and Lybanon, M. Algorithm 515: generation of a vector from the lexicographical index ACM Transactions on Mathematical Software (TOMS), 1977 vol. 3, n. 2, pages 180–182 ACM

[AW] Adams, M.D. and Wise, D.S. Fast additions on masked integers ACM SIGPLAN Notices, 2006 vol. 41, n.5, pages 39–45 ACM http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1801&rep=rep1&type=pdf

[HIK11] Handbook of Product Graphs, R. Hammack, W. Imrich, S. Klavzar, CRC press, 2011

[FMDecb] Fabien de Montgolfier http://www.liafa.jussieu.fr/~fm/algos/index.html

[HabibViennot1999b] Michel Habib, Christiphe Paul, Laurent Viennot Partition refinement techniques: An interesting algorithmic tool kit International Journal of Foundations of Computer Science vol. 10 n2 pp.147–170, 1999

[CapHabMont02b] C. Capelle, M. Habib et F. de Montgolfier Graph decomposition and Factorising Permutations Discrete Mathematics and Theoretical Computer Sciences, vol 5 no. 1 , 2002.

[CHZ02] F. Harary, E. Loukakis, C. Tsouros The geodetic number of a graph Mathematical and computer modelling vol. 17 n11 pp.89–95, 1993

[NikolopoulosPalios07] Nikolopoulos, S.D. and Palios, L. Detecting holes and antiholes in graphs Algorithmica, 2007 Vol. 47, number 2, pages 119–138 http://www.cs.uoi.gr/~stavros/C-Papers/C-2004-SODA.pdf

[KRG96b] S. Klavzar, A. Rajapakse, and I. Gutman. The Szeged and the Wiener index of graphs. *Applied Mathematics Letters*, 9(5):45–49, 1996.

[GYLL93c] I. Gutman, Y.-N. Yeh, S.-L. Lee, and Y.-L. Luo. Some recent results in the theory of the Wiener number. *Indian Journal of Chemistry*, 32A:651–661, 1993.

[CCL12] N. Cohen, D. Coudert, and A. Lancin. Exact and approximate algorithms for computing the hyperbolicity of large-scale graphs. Research Report RR-8074, Sep. 2012. [http://hal.inria.fr/hal-00735481].

[FIV12] H. Fournier, A. Ismail, and A. Vigneron. Computing the Gromov hyperbolicity of a discrete metric space. ArXiv, Tech. Rep. arXiv:1210.3323, Oct. 2012. [http://arxiv.org/abs/1210.3323].

[Gromov87] M. Gromov. Hyperbolic groups. Essays in Group Theory, 8:75–263, 1987.

[Gordon10] Computing Tutte Polynomials. Gary Haggard, David J. Pearce and Gordon Royle. In ACM Transactions on Mathematical Software, Volume 37(3), article 24, 2010. Preprint: http://homepages.ecs.vuw.ac.nz/~djp/files/TOMS10.pdf

PYTHON MODULE INDEX

c

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W