Graph theory package for Giac/Xcas

Reference manual

August 2018

DRAFT

Table of contents

Introduction

This document contains an overview of the graph theory commands built in the Giac computation kernel, including the calling syntax, detailed description and practical examples for each command.

Chapter 1

Constructing graphs

1.1. General graphs

The commands `graph` and `digraph` are used for constructing general graphs.

1.1.1. Creating undirected graphs

The command `graph` accepts between one and three mandatory arguments, each of them being one of the following structural elements of the resulting graph:

- the number or list of vertices (a vertex may be any atomic object, such as an integer, a symbol or a string); it must be the first argument if used,
- the set of edges (each edge is a list containing two vertices), a permutation, a trail of edges or a sequence of trails; it can be either the first or the second argument if used,
- the adjacency or weight matrix.

Additionally, some of the following options may be appended to the sequence of arguments:

- `directed = true` or `false`,
- `weighted = true` or `false`,
- `color =` an integer or a list of integers representing color(s) of the vertices,
- `coordinates =` a list of vertex 2D or 3D coordinates.

The `graph` command may also be called by passing a string, representing the name of a special graph, as its only argument. In that case the corresponding graph will be constructed and returned. The supported graphs and their names are listed below.

1. 2nd Blanuša snark: `blanusa`
2. Clebsch graph: `clebsch`
3. Coxeter graph: `coxeter`
4. Desargues graph: `desargues`
5. Dodecahedral graph: `dodecahedron`
6. Dürer graph: `durer`
7. Dyck graph: `dyck`
8. Grinberg graph: `grinberg`
9. Grotzsch graph: `grotzsch`
10. Harries graph: `harries`

11. Harries–Wong graph: `harries-wong`
12. Heawood graph: `heawood`
13. Herschel graph: `herschel`
14. Icosahedral graph: `icosahedron`
15. Levi graph: `levi`
16. Ljubljana graph: `ljubljana`
17. McGee graph: `mcgee`
18. Möbius–Kantor graph: `mobius-kantor`
19. Nauru graph: `nauru`
20. Octahedral graph: `octahedron`
21. Pappus graph: `pappus`
22. Petersen graph: `petersen`
23. Robertson graph: `robertson`
24. Truncated icosahedral graph: `soccerball`
25. Shrikhande graph: `shrikhande`
26. Tetrahedral graph: `tehtrahedron`

1.1.2. Creating directed graphs

The `digraph` command is used for creating directed graphs, although it is also possible with the `graph` command by specifying the option `directed=true`. Actually, calling `digraph` is the same as calling `graph` with that option appended to the sequence of arguments. However, creating special graphs is not supported by `digraph` since they are all undirected. Edges in directed graphs are called *arcs*. Edges and arcs are different structures: an edge is represented by a two-element set containing its endpoints, while an arc is represented by the ordered pairs of its endpoints.

The following series of examples demonstrates the various possibilities when using `graph` and `digraph` commands.

1.1.3. Creating vertices

A graph consisting only of vertices and no edges can be created simply by providing the number of vertices or the list of vertex labels.

```
> graph(5)
```

an undirected unweighted graph with 5 vertices and 0 edges

```
> graph([a,b,c])
```

an undirected unweighted graph with 3 vertices and 0 edges

The commands that return graphs often need to generate vertex labels. In these cases ordinal integers are used, which are 0-based in Xcas mode and 1-based in Maple mode. Examples throughout this manual are made using the default, Xcas mode.

1.1.4. Creating edges and arcs

Edges/arcs must be specified inside a set so that it can be distinguished from a (adjacency or weight) matrix. If only a set of edges/arcs is specified, the vertices needed to establish these will be created automatically. Note that, when constructing a directed graph, the order of the vertices in an arc matters; in undirected graphs it is not meaningful.

```
> graph(%{[a,b],[b,c],[a,c]%})
```

an undirected unweighted graph with 3 vertices and 3 edges

Edge weights may also be specified.

```
> graph(%{[[a,b],2],[[b,c],2.3],[[c,a],3/2]%})
```

an undirected weighted graph with 3 vertices and 3 edges

If the graph contains isolated vertices (not connected to any other vertex) or a particular order of vertices is desired, the list of vertices has to be specified first.

```
> graph([d,b,c,a],%{[a,b],[b,c],[a,c]%})
```

an undirected unweighted graph with 4 vertices and 3 edges

1.1.5. Creating paths and trails

A directed graph can also be created from a list of n vertices and a permutation of order n. The resulting graph consists of a single directed path with the vertices ordered according to the permutation.

```
> graph([a,b,c,d],[1,2,3,0])
```

a directed unweighted graph with 4 vertices and 3 arcs

Alternatively, one may specify edges as a trail.

```
> digraph([a,b,c,d],trail(b,c,d,a))
```

a directed unweighted graph with 4 vertices and 3 arcs

Using trails is also possible when creating undirected graphs. Also, some vertices in a trail may be repeated, which is not allowed in a path.

```
> graph([a,b,c,d],trail(b,c,d,a,c))
```

an undirected unweighted graph with 4 vertices and 4 edges

There is also the possibility of specifying several trails in a sequence, which is useful for designing more complex graphs.

```
> graph(trail(1,2,3,4,2),trail(3,5,6,7,5,4))
```

an undirected unweighted graph with 7 vertices and 9 edges

1.1.6. Specifying adjacency or weight matrix

A graph can be created from a single square matrix $A = [a_{ij}]_n$ of order n. If it contains only ones and zeros and has zeros on its diagonal, it is assumed to be the adjacency matrix for the desired graph. Otherwise, if an element outside the set $\{0, 1\}$ is encountered, it is assumed that the matrix of edge weights is passed as input, causing the resulting graph to be weighted accordingly. In each case, exactly n vertices will be created and i-th and j-th vertex will be connected iff $a_{ij} \neq 0$. If the matrix is symmetric, the resulting graph will be undirected, otherwise it will be directed.

```
> graph([[0,1,1,0],[1,0,0,1],[1,0,0,0],[0,1,0,0]])
```

an undirected unweighted graph with 4 vertices and 3 edges

```
> graph([[0,1.0,2.3,0],[4,0,0,3.1],[0,0,0,0],[0,0,0,0]])
```

a directed weighted graph with 4 vertices and 4 arcs

List of vertex labels can be specified before the matrix.

```
> graph([a,b,c,d],[[0,1,1,0],[1,0,0,1],[1,0,0,0],[0,1,0,0]])
```

an undirected unweighted graph with 4 vertices and 3 edges

When creating a weighted graph, one can first specify the list of n vertices and the set of edges, followed by a square matrix A of order n. Then for every edge $\{i, j\}$ or arc (i, j) the element a_{ij} of A is assigned as its weight. Other elements of A are ignored.

```
> digraph([a,b,c],%{[a,b],[b,c],[a,c]%}, [[0,1,2],[3,0,4],[5,6,0]])
```

a directed weighted graph with 3 vertices and 3 arcs

When a special graph is desired, one just needs to pass its name to the `graph` command. An undirected unweighted graph will be returned.

```
> graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

1.2. PROMOTING TO DIRECTED AND UNDIRECTED GRAPHS

1.2.1. Converting edges to pairs of arcs

To promote an existing undirected graph to a directed one, use the command `make_directed`.

`make_directed` is called with one or two arguments, an undirected graph $G(V, E)$ and optionally a square matrix of order $|V|$. Every edge $\{i, j\} \in E$ is replaced with the pair of arcs (i, j) and (j, i). If matrix A is specified, a_{ij} and a_{ji} are assigned as weights of these arcs, respectively. Thus a directed (and possibly weighted) graph is created and returned.

```
> make_directed(cycle_graph(4))
```

C4: a directed unweighted graph with 4 vertices and 8 arcs

```
> make_directed(cycle_graph(4), [[0,0,0,1],[2,0,1,3],[0,1,0,4],[5,0,4,0]])
```

C4: a directed weighted graph with 4 vertices and 8 arcs

1.2.2. Assigning weight matrix to an unweighted graph

To promote an existing unweighted graph to a weighted one, use the command `make_weighted`.

`make_weighted` accepts one or two arguments, an unweighted graph $G(V, E)$ and optionally a square matrix A of order $|V|$. If the matrix specification is omitted, a square matrix of ones is assumed. Then a copy of G is returned where each edge/arc $(i, j) \in E$ gets a_{ij} assigned as its weight. If G is an undirected graph, it is assumed that A is symmetric.

```
> make_weighted(graph(%{[1,2],[2,3],[3,1]%}), [[0,2,3],[2,0,1],[3,1,0]])
```

an undirected weighted graph with 3 vertices and 3 edges

1.3. Cycle and path graphs

1.3.1. Cycle graphs

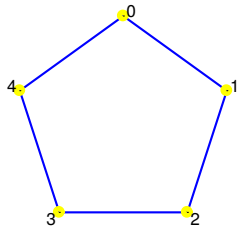
The command `cycle_graph` is used for constructing a cycle graph of arbitrary order.

`cycle_graph` accepts a positive integer n or a list of distinct vertices as its only argument and returns the graph consisting of a single cycle through the specified vertices in the given order. If n is specified it is assumed to be the desired number of vertices, in which case they will be created and labeled with the first n integers (starting from 0 in Xcas mode and from 1 in Maple mode). The resulting graph will be given the name `Cn`, for example `C4` for $n=4$.

```
> C5:=cycle_graph(5)
```

C5: an undirected unweighted graph with 5 vertices and 5 edges

```
> draw_graph(C5)
```




```
> cycle_graph(["a","b","c","d","e"])
```

C5: an undirected unweighted graph with 5 vertices and 5 edges

1.3.2. Path graphs

The command `path_graph` is used for constructing a path graph of arbitrary length.

`path_graph` accepts a positive integer n or a list of distinct vertices as its only argument and returns a graph consisting of a single path through the specified vertices in the given order. If n is specified it is assumed to be the desired number of vertices, in which case they will be created and labeled with the first n integers (starting from 0 in Xcas mode resp. from 1 in Maple mode).

Note that a path cannot intersect itself. Paths that are allowed to cross themselves are called *trails* (see the command `trail`).

```
> path_graph(5)
```

an undirected unweighted graph with 5 vertices and 4 edges

```
> path_graph(["a","b","c","d","e"])
```

an undirected unweighted graph with 5 vertices and 4 edges

1.3.3. Trails of edges

If the dummy command `trail` is called with a sequence of vertices as arguments, it returns the symbolic expression representing the trail of edges through the specified vertices. The resulting symbolic object is recognizable by `graph` and `digraph` commands. Note that a trail may cross itself (some vertices may be repeated in the given sequence).

Any trail is easily converted to the corresponding list of edges by calling the `trail2edges` command, which accepts the trail as its only argument.

```
> T:=trail(1,2,3,4,2):; graph(T)
```

Done, an undirected unweighted graph with 4 vertices and 4 edges

```
> trail2edges(T)
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 2 \end{pmatrix}$$

1.4. COMPLETE GRAPHS

1.4.1. Complete graphs with multiple vertex partitions

The command `complete_graph` is used for construction of complete (multipartite) graphs.

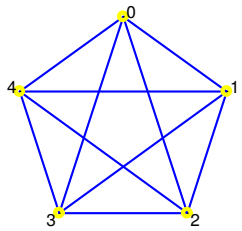
If `complete_graph` is called with a single argument, a positive integer n or a list of distinct vertices, it returns the complete graph with the specified vertices. If integer n is specified, it is assumed that it is the desired number of vertices and they will be created and labeled with the first n integers (starting from 0 in Xcas mode and from 1 in Maple mode).

If a sequence of positive integers $n_1, n_2, ..., n_k$ is passed as argument, `complete_graph` returns the complete multipartite graph with partitions of size $n_1, n_2, ..., n_k$.

```
> K5:=complete_graph(5)
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> draw_graph(K5)
```




```
> K3:=complete_graph([a,b,c])
```

an undirected unweighted graph with 3 vertices and 3 edges

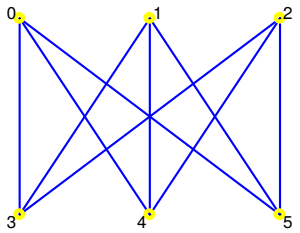
```
> edges(K3)
```

$$\{[a,b],[a,c],[b,c]\}$$

```
> K33:=complete_graph(3,3)
```

an undirected unweighted graph with 6 vertices and 9 edges

```
> draw_graph(K33)
```



1.4.2. Complete trees

To construct the complete binary tree of depth n, use the command `complete_binary_tree` which accepts n (a positive integer) as its only argument.

```
> complete_binary_tree(2)
```

an undirected unweighted graph with 7 vertices and 6 edges

To construct the complete k-ary tree of the specified depth use the command `complete_kary_tree`.

`complete_kary_tree` accepts k and n (positive integers) as its arguments and returns the complete k-ary tree of depth n. For example, to get a ternary tree with two levels, input:

```
> complete_kary_tree(3,2)
```

an undirected unweighted graph with 13 vertices and 12 edges

1.5. Sequence graphs

1.5.1. Creating graphs from degree sequences

The command `sequence_graph` is used for constructing the graph from its degree sequence.

`sequence_graph` accepts a list L of positive integers as its only argument and, if L represents a graphic sequence, the corresponding graph G with $|L|$ vertices is returned. If the argument is not a graphic sequence, an error is returned.

```
> sequence_graph([3,2,4,2,3,4,5,7])
```

an undirected unweighted graph with 8 vertices and 15 edges

The graph G is constructed in $O(|L|^2 \log |L|)$ time by using the algorithm of Havel and Hakimi.

1.5.2. Validating graphic sequences

The command `is_graphic_sequence` is used to check whether a list of integers represents the degree sequence of some graph or not.

`is_graphic_sequence` accepts a list L of positive integers as its only argument and returns `true` if there exists a graph $G(V, E)$ with degree sequence $\{\deg v : v \in V\}$ equal to L and `false` otherwise. The algorithm, which has the complexity $O(|L|^2)$, is based on the theorem of Erdős and Gallai.

```
> is_graphic_sequence([3,2,4,2,3,4,5,7])
```

true

1.6. Intersection graphs

1.6.1. Interval graphs

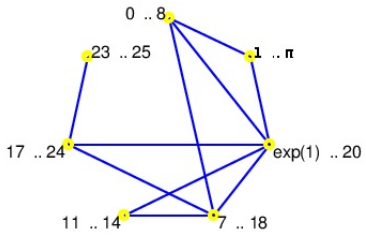
The command `interval_graph` is used for construction of interval graphs.

`interval_graph` accepts a sequence or list of real-line intervals as its argument and returns an undirected unweighted graph with these intervals as vertices (the string representations of the intervals are used as labels), each two of them being connected with an edge if and only if the corresponding intervals intersect.

```
> G:=interval_graph(0..8,1..pi,exp(1)..20,7..18,11..14,17..24,23..25)
```

an undirected unweighted graph with 7 vertices and 10 edges

```
> draw_graph(G)
```



1.6.2. Kneser graphs

The commands `kneser_graph` and `odd_graph` are used for construction of Kneser graphs.

`kneser_graph` accepts two positive integers $n \leq 20$ and k as its arguments and returns the Kneser graph $K(n,k)$. The latter is obtained by setting all k-subsets of a set of n elements as vertices and connecting each two of them if and only if the corresponding sets are disjoint. Therefore, each Kneser graph is the complement of the corresponding intersection graph on the same collection of subsets.

Kneser graphs can get exceedingly complex even for relatively small values of n and k. Note that the number of vertices in $K(n,k)$ is equal to $\binom{n}{k}$.

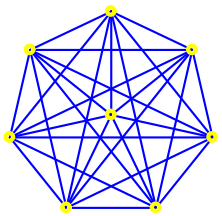
```
> kneser_graph(5,2)
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> G:=kneser_graph(8,1)
```

an undirected unweighted graph with 8 vertices and 28 edges

```
> draw_graph(G,spring,labels=false)
```



The command `odd_graph` is used for creating so-called *odd* graphs, which are Kneser graphs with parameters $n = 2d+1$ and $k = d$ for $d \geqslant 1$.

`odd_graph` accepts a positive integer $d \leq 8$ as its only argument and returns d-th odd graph $K(2d+1, d)$. Note that the odd graphs with $d > 8$ will not be constructed as they are too big to handle.

```
> odd_graph(3)
```

an undirected unweighted graph with 10 vertices and 15 edges

1.7. SPECIAL GRAPHS

1.7.1. Hypercube graphs

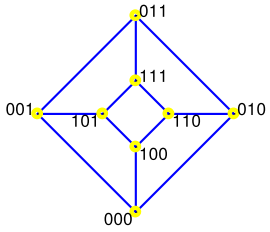
The command `hypercube_graph` is used for creating hypercube graphs.

`hypercube_graph` accepts a positive integer n as its only argument and returns the hypercube graph of dimension n on 2^n vertices. The vertex labels are strings of binary digits of length n. Two vertices are joined by an edge if and only if their labels differ in exactly one character. The hypercube graph for $n=2$ is a square and for $n=3$ it is a cube.

```
> H:=hypercube_graph(3)
```

an undirected unweighted graph with 8 vertices and 12 edges

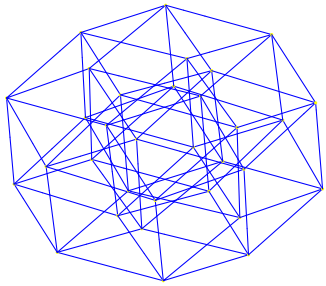
```
> draw_graph(H,planar)
```




```
> H:=hypercube_graph(5)
```

an undirected unweighted graph with 32 vertices and 80 edges

```
> draw_graph(H,plot3d,labels=false)
```



1.7.2. Star graphs

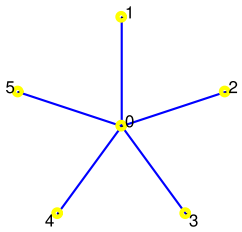
The command `star_graph` is used for creating star graphs.

`star_graph` accepts a positive integer n as its only argument and returns the star graph with $n+1$ vertices, which is equal to the complete bipartite graph `complete_graph(1,n)` i.e. a n-ary tree with one level.

```
> G:=star_graph(5)
```

an undirected unweighted graph with 6 vertices and 5 edges

```
> draw_graph(G)
```



1.7.3. Wheel graphs

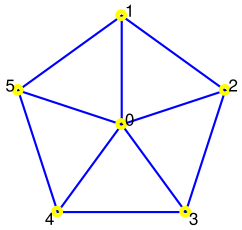
The command `wheel_graph` is used for creating wheel graphs.

`wheel_graph` accepts a positive integer n as its only argument and returns the wheel graph with $n+1$ vertices.

```
> G:=wheel_graph(5)
```

an undirected unweighted graph with 6 vertices and 10 edges

```
> draw_graph(G)
```



1.7.4. Web graphs

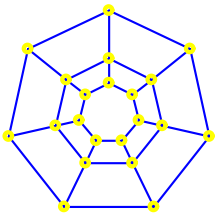
The command `web_graph` is used for creating web graphs.

`web_graph` accepts two positive integers a and b as its arguments and returns the web graph with parameters a and b, namely the Cartesian product of `cycle_graph(a)` and `path_graph(b)`.

```
> G:=web_graph(7,3)
```

an undirected unweighted graph with 21 vertices and 35 edges

```
> draw_graph(G,labels=false)
```



1.7.5. Prism graphs

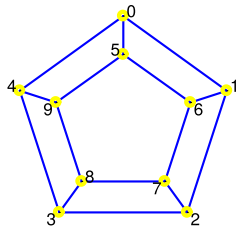
The command `prism_graph` is used for creating prism graphs.

`prism_graph` accepts a positive integer n as its only argument and returns the prism graph with parameter n, namely `petersen_graph(n,1)`.

```
> G:=prism_graph(5)
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> draw_graph(G)
```



1.7.6. Antiprism graphs

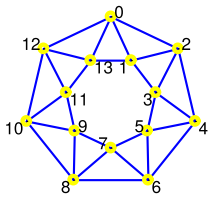
The command `antiprism_graph` is used for creating antiprism graphs.

`antiprism_graph` accepts a positive integer n as its only argument and returns the antiprism graph with parameter n, which is constructed from two concentric cycles of n vertices by joining each vertex of the inner to two adjacent nodes of the outer cycle.

```
> G:=antiprism_graph(7)
```

an undirected unweighted graph with 14 vertices and 28 edges

```
> draw_graph(G)
```



1.7.7. Grid graphs

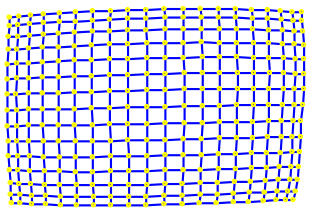
The command `grid_graph` resp. `torus_grid_graph` is used for creating rectangular resp. torus grid graphs.

`grid_graph` accepts two positive integers m and n as its arguments and returns the m by n grid on $m \cdot n$ vertices, namely the Cartesian product of `path_graph(m)` and `path_graph(n)`.

```
> G:=grid_graph(15,20)
```

an undirected unweighted graph with 300 vertices and 565 edges

```
> draw_graph(G,spring)
```

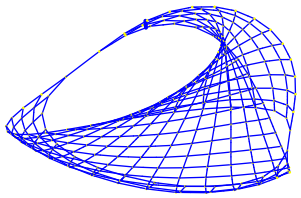


For example, connecting vertices in the opposite corners of the generated grid yields a grid-like graph with no corners.

```
> G:=add_edge(G,[["14:0","0:19"],["0:0","14:19"]])
```

an undirected unweighted graph with 300 vertices and 567 edges

```
> draw_graph(G,plot3d)
```



In the next example, the Möbius strip is constructed by connecting the vertices in the opposite sides of a narrow grid graph.

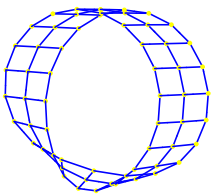
```
> G:=grid_graph(20,3)
```

an undirected unweighted graph with 60 vertices and 97 edges

```
> G:=add_edge(G,[["0:0","19:2"],["0:1","19:1"],["0:2","19:0"]])
```

an undirected unweighted graph with 60 vertices and 100 edges

```
> draw_graph(G,plot3d,labels=false)
```

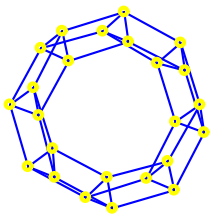


`torus_grid_graph` accepts two positive integers m and n as its arguments and returns the m by n torus grid on mn vertices, namely the Cartesian product of `cycle_graph(m)` and `cycle_graph(n)`.

```
> G:=torus_grid_graph(8,3)
```

an undirected unweighted graph with 24 vertices and 48 edges

```
> draw_graph(G,spring,labels=false)
```



1.7.8. Sierpiński graphs

The command `sierpinski_graph` is used for creating Sierpiński-type graphs S_k^n and ST_k^n [17].

`sierpinski_graph` accepts two positive integers n and k as its arguments (and optionally the symbol `triangle` as the third argument) and returns the Sierpiński (triangle) graph with parameters n and k.

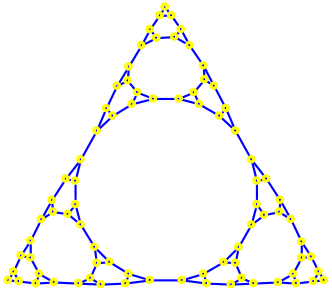
The Sierpiński triangle graph ST_k^n is obtained by contracting all non-clique edges in S_k^n. To detect such edges the variant of the algorithm by BRON and KERBOSCH, developed by TOMITA et al. in [31], is used, which can be time consuming for $n > 6$.

In particular, ST_3^n is the well-known Sierpiński sieve[1.1] graph of order n.

```
> S:=sierpinski_graph(4,3)
```

an undirected unweighted graph with 81 vertices and 120 edges

```
> draw_graph(S,spring)
```



1.1. https://en.wikipedia.org/wiki/Sierpinski_triangle

```
> sierpinski_graph(4,3,triangle)
```

an undirected unweighted graph with 42 vertices and 81 edges

```
> sierpinski_graph(5,3,triangle)
```

an undirected unweighted graph with 123 vertices and 243 edges

A drawing of the graph produced by the last command line is shown in Figure 3.1.

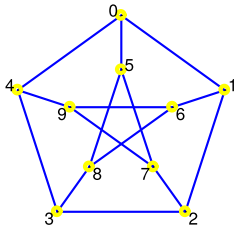
1.7.9. Generalized Petersen graphs

The command `petersen_graph` is used for creating generalized Petersen graphs $P(n, k)$.

`petersen_graph` accepts two arguments, n and k (positive integers). The second argument may be omitted, in which case $k = 2$ is assumed. The graph $P(n, k)$, which is returned, is a connected cubic graph consisting of—in Schläfli notation—an inner star polygon $\{n, k\}$ and an outer regular polygon $\{n\}$ such that the n pairs of corresponding vertices in inner and outer polygons are connected with edges. For $k = 1$ the prism graph of order n is obtained.

The well-known Petersen graph is equal to the generalized Petersen graph $P(5, 2)$. It can also be constructed by calling `graph("petersen")`.

```
> draw_graph(graph("petersen"))
```



To obtain the dodecahedral graph $P(10, 2)$, input:

```
> petersen_graph(10)
```

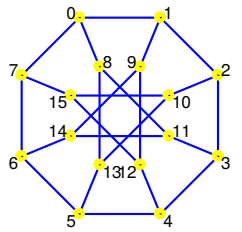
an undirected unweighted graph with 20 vertices and 30 edges

To obtain Möbius–Kantor graph $P(8, 3)$, input:

```
> G:=petersen_graph(8,3)
```

an undirected unweighted graph with 16 vertices and 24 edges

```
> draw_graph(G)
```



Note that Desargues, Dürer and Nauru graphs are isomorphic to the generalized Petersen graphs $P(10, 3)$, $P(6, 2)$ and $P(12, 5)$, respectively.

1.7.10. LCF graphs

The command `lcf_graph` is used for constructing a cubic Hamiltonian graph from its LCF notation[1,2].

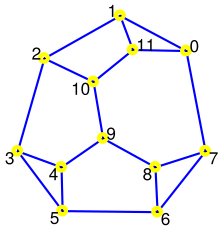
`lcf_graph` takes one or two arguments, a list L of nonzero integers, called *jumps*, and optionally a positive integer n, called *exponent* (by default, $n=1$). The command returns the graph on $n\,|L|$ vertices obtained by iterating the sequence of jumps n times.

For example, the following command line creates the Frucht graph.

```
> F:=lcf_graph([-5,-2,-4,2,5,-2,2,5,-2,-5,4,2])
```

an undirected unweighted graph with 12 vertices and 18 edges

```
> draw_graph(F,planar)
```

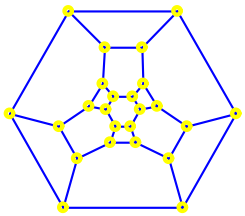


The following command line constructs the truncated octahedral graph.

```
> G:=lcf_graph([3,-7,7,-3],6)
```

an undirected unweighted graph with 24 vertices and 36 edges

```
> draw_graph(G,planar,labels=false)
```



1.8. ISOMORPHIC COPIES OF GRAPHS

1.8.1. Creating an isomorphic copy of a graph

To create an isomorphic copy of a graph by applying a permutation to the ordering of vertices, use the `isomorphic_copy` command.

`isomorphic_copy` accepts two arguments, a graph $G(V,E)$ and a permutation σ of order $|V|$, and returns the copy of graph G with vertices rearranged according to σ. The complexity of the algorithm is $O(|V|+|E|)$.

```
> P:=path_graph([1,2,3,4,5])
```

an undirected unweighted graph with 5 vertices and 4 edges

```
> G:=isomorphic_copy(P,randperm(5))
```

an undirected unweighted graph with 5 vertices and 4 edges

```
> vertices(G)
```

$[3,4,2,1,5]$

```
> is_isomorphic(P,G)
```

1.2. For the details about LCF notation, see https://en.wikipedia.org/wiki/LCF_notation.

true

1.8.2. Permuting graph vertices

To create an isomorphic copy of a graph by providing the reordered list of vertex labels, use the command `permute_vertices`.

`permute_vertices` accepts two arguments, a graph $G(V, E)$ and a list L of length $|V|$ containing all vertices from V in a certain order, and returns a copy of G with vertices rearranged as specified by L. The complexity of the algorithm is $O(|V| + |E|)$.

```
> G:=permute_vertices(path_graph([a,b,c,d]),[b,d,a,c])
```

an undirected unweighted graph with 4 vertices and 3 edges

```
> vertices(G)
```

$[b, d, a, c]$

1.8.3. Relabeling graph vertices

To relabel the vertices of a graph without changing their order, use the command `relabel_vertices`.

`relabel_vertices` accepts two arguments, a graph $G(V, E)$ and a list L of vertex labels, and returns the copy of G with vertices relabeled with labels from L. The complexity of the algorithm is $O(|V|)$.

```
> P:=path_graph([1,2,3,4])
```

an undirected unweighted graph with 4 vertices and 3 edges

```
> edges(P)
```

$\{[1, 2], [2, 3], [3, 4]\}$

```
> G:=relabel_vertices(P,[a,b,c,d])
```

an undirected unweighted graph with 4 vertices and 3 edges

```
> edges(G)
```

$\{[a, b], [b, c], [c, d]\}$

1.9. Subgraphs

1.9.1. Extracting subgraphs

To extract the subgraph of $G(V, E)$ formed by edges from $L \subset E$, use the command `subgraph` which accepts two arguments: G and L.

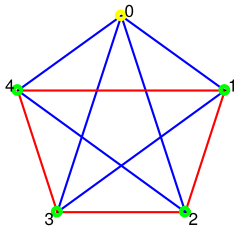
```
> K5:=complete_graph(5)
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> S:=subgraph(K5,[[1,2],[2,3],[3,4],[4,1]])
```

an undirected unweighted graph with 4 vertices and 4 edges

```
> draw_graph(highlight_subgraph(K5,S))
```



1.9.2. Induced subgraphs

To obtain the subgraph of G induced by set of vertices $L \subset V$, use the command `induced_subgraph`.

`induced_subgraph` accepts two arguments G and L and returns the subgraph of G formed by all edges in E which have endpoints in L.

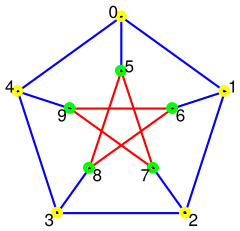
```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> S:=induced_subgraph(G,[5,6,7,8,9])
```

an undirected unweighted graph with 5 vertices and 5 edges

```
> draw_graph(highlight_subgraph(G,S))
```



1.9.3. Underlying graphs

For every graph $G(V, E)$ there is an undirected and unweighted graph $U(V, E')$, called the *underlying graph* of G, where E' is obtained from E by dropping edge directions.

To construct U use the command `underlying_graph` which accepts G as its only argument. The result is obtained by copying G and "forgetting" the directions of edges as well as their weights. The complexity of the algorithm is $O(|V| + |E|)$.

```
> G:=digraph(%{[[1,2],6],[[2,3],4],[[3,1],5],[[3,2],7]%})
```

a directed weighted graph with 3 vertices and 4 arcs

```
> U:=underlying_graph(G)
```

an undirected unweighted graph with 3 vertices and 3 edges

```
> edges(U)
```

$\{[1, 2], [1, 3], [2, 3]\}$

1.10. OPERATIONS ON GRAPHS

1.10.1. Graph complement

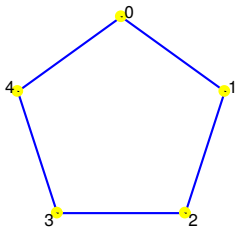
The command `graph_complement` is used for constructing complements of graphs.

`graph_complement` accepts a graph $G(V, E)$ as its only argument and returns the complement $G^c(V, E^c)$ of G, where E^c is the largest set containing only edges/arcs not present in G. The complexity of the algorithm is $O(|V|^2)$.

```
> C5:=cycle_graph(5)
```

C5: an undirected unweighted graph with 5 vertices and 5 edges

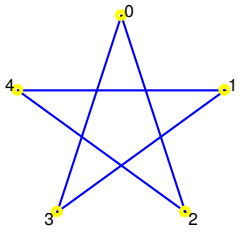
```
> draw_graph(C5)
```




```
> G:=graph_complement(C5)
```

an undirected unweighted graph with 5 vertices and 5 edges

```
> draw_graph(G)
```



1.10.2. Seidel switching

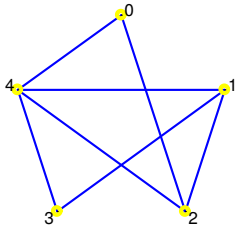
The command `seidel_switch` is used for Seidel switching in graphs.

`seidel_switch` accepts two arguments, an undirected and unweighted graph $G(V, E)$ and a list of vertices $L \subset V$. The result is a copy of G in which, for each vertex $v \in L$, its neighbors become its non-neighbors and vice versa.

```
> S:=seidel_switch(cycle_graph(5),[1,2])
```

an undirected unweighted graph with 5 vertices and 7 edges

```
> draw_graph(S)
```



1.10.3. Transposing graphs

The command `reverse_graph` is used for reversing arc directions in digraphs.

`reverse_graph` accepts a graph $G(V, E)$ as its only argument and returns the reverse graph $G^T(V, E')$ of G where $E' = \{(j, i) : (i, j) \in E\}$, i.e. returns the copy of G with the directions of all edges reversed.

Note that `reverse_graph` is defined for both directed and undirected graphs, but gives meaningful results only for directed graphs.

G^T is also called the *transpose graph* of G because adjacency matrices of G and G^T are transposes of each other (hence the notation).

```
> G:=digraph(6, %{[1,2],[2,3],[2,4],[4,5]%})
```

a directed unweighted graph with 6 vertices and 4 arcs

```
> GT:=reverse_graph(G)
```

a directed unweighted graph with 6 vertices and 4 arcs

```
> edges(GT)
```

$\{[2,1],[3,2],[4,2],[5,4]\}$

1.10.4. Union of graphs

The command `graph_union` is used for constructing union of two or more graphs.

`graph_union` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k = 1, 2, ..., n$ as its argument and returns the graph $G(V, E)$ where $V = V_1 \cup V_2 \cup \cdots \cup V_k$ and $E = E_1 \cup E_2 \cup \cdots \cup E_k$.

```
> G1:=graph([1,2,3],%{[1,2],[2,3]%})
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> G2:=graph([1,2,3],%{[3,1],[2,3]%})
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> G:=graph_union(G1,G2)
```

an undirected unweighted graph with 3 vertices and 3 edges

```
> edges(G)
```

$\{[1,2],[1,3],[2,3]\}$

1.10.5. Disjoint union of graphs

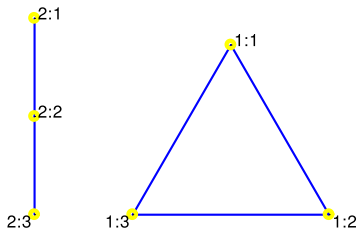
To construct disjoint union of graphs use the command `disjoint_union`.

`disjoint_union` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k = 1, 2, ..., n$ as its argument and returns the graph obtained by labeling all vertices with strings `k:v` where $v \in V_k$ and all edges with strings `k:e` where $e \in E_k$ and calling the `graph_union` command subsequently. As all vertices and edges are labeled differently, it follows $|V| = \sum_{k=1}^{n} |V_k|$ and $|E| = \sum_{k=1}^{n} |E_k|$.

```
> G:=disjoint_union(cycle_graph([1,2,3]),path_graph([1,2,3]))
```

an undirected unweighted graph with 6 vertices and 5 edges

```
> draw_graph(G)
```



1.10.6. Joining two graphs

The command `graph_join` is used for joining graphs.

`graph_join` accepts two graphs G and H as its arguments and returns the graph $G+H$ which is obtained by connecting all the vertices of G to all vertices of H. The vertex labels in the resulting graph are strings of the form `1:u` and `2:v` where u is a vertex in G and v is a vertex in H.

```
> G:=path_graph(2)
```

an undirected unweighted graph with 2 vertices and 1 edge

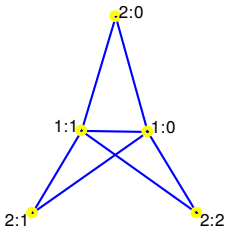
```
> H:=graph(3)
```

an undirected unweighted graph with 3 vertices and 0 edges

```
> GH:=graph_join(G,H)
```

an undirected unweighted graph with 5 vertices and 7 edges

```
> draw_graph(GH,spring)
```



1.10.7. Power graphs

The command `graph_power` is used for constructing the powers of a graph.

`graph_power` accepts two arguments, a graph $G(V,E)$ and a positive integer k, and returns the k-th power G^k of G with vertices V such that $v,w \in V$ are connected with an edge if and only if there exists a path of length at most k in G.

The graph G^k is constructed from its adjacency matrix A_k which is obtained by adding powers of the adjacency matrix A of G:

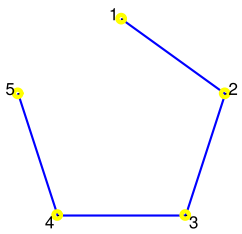
$$A_k = \sum_{i=1}^{k} A^k.$$

The above sum is obtained by assigning $A_k \leftarrow A$ and repeating the instruction $A_k \leftarrow (A_k + I)\, A$ for $k-1$ times, so exactly k matrix multiplications are required.

```
> G:=graph(trail(1,2,3,4,5))
```

an undirected unweighted graph with 5 vertices and 4 edges

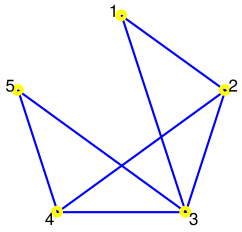
```
> draw_graph(G,circle)
```




```
> P2:=graph_power(G,2)
```

an undirected unweighted graph with 5 vertices and 7 edges

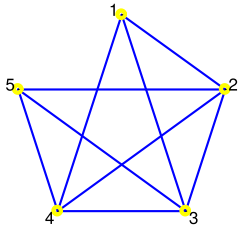
```
> draw_graph(P2,circle)
```




```
> P3:=graph_power(G,3)
```

an undirected unweighted graph with 5 vertices and 9 edges

```
> draw_graph(P3,circle)
```



1.10.8. Graph products

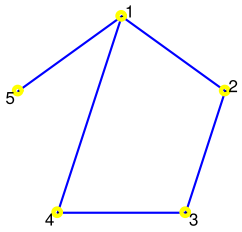
There are two distinct operations for computing the product of two graphs: Cartesian product and tensor product. These operations are available in Giac as the commands `cartesian_product` and `tensor_product`, respectively.

The command `cartesian_product` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k=1,2,...,n$ as its argument and returns the Cartesian product $G_1 \times G_2 \times \cdots \times G_n$ of the input graphs. The Cartesian product $G(V,E)=G_1 \times G_2$ is the graph with list of vertices $V=V_1 \times V_2$, labeled with strings `v1:v2` where $v_1 \in V_1$ and $v_2 \in V_2$, such that (`u1:v1`,`u2:v2`)$\in E$ if and only if u_1 is adjacent to u_2 and $v_1=v_2$ **or** $u_1=u_2$ and v_1 is adjacent to v_2.

```
> G1:=graph(trail(1,2,3,4,1,5))
```

an undirected unweighted graph with 5 vertices and 5 edges

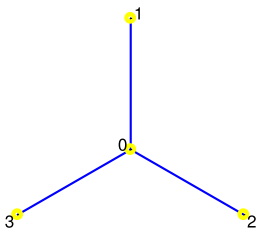
```
> draw_graph(G1,circle)
```




```
> G2:=star_graph(3)
```

an undirected unweighted graph with 4 vertices and 3 edges

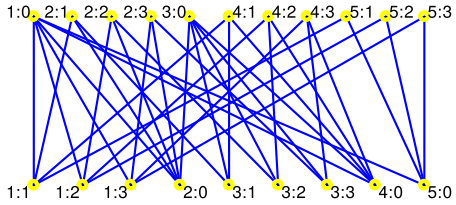
```
> draw_graph(G2,circle=[1,2,3])
```



```
> G:=cartesian_product(G1,G2)
```

an undirected unweighted graph with 20 vertices and 35 edges

```
> draw_graph(G)
```

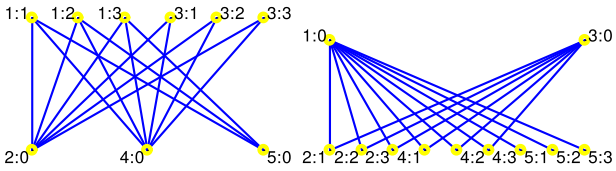


The command `tensor_product` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k = 1, 2, ..., n$ as its argument and returns the tensor product $G_1 \times G_2 \times \cdots \times G_n$ of the input graphs. The tensor product $G(V, E) = G_1 \times G_2$ is the graph with list of vertices $V = V_1 \times V_2$, labeled with strings `v1:v2` where $v_1 \in V_1$ and $v_2 \in V_2$, such that (`u1:v1`,`u2:v2`)$\in E$ if and only if u_1 is adjacent to u_2 **and** v_1 is adjacent to v_2.

```
> T:=tensor_product(G1,G2)
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> draw_graph(T)
```



1.10.9. Transitive closure graph

The command `transitive_closure` is used for constructing the transitive closure graph of the given graph. Transitive closure of a (directed) graph $G(V, E)$ is the graph $T(V, E')$ in which two vertices $v, w \in V$ are connected by an edge (arc) $e = vw \in E'$ if and only if there exists a (directed) path from v to w in G.

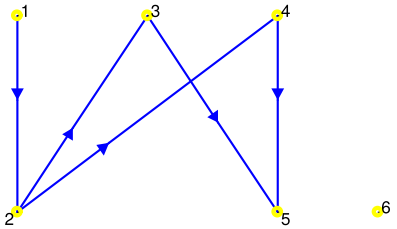
`transitive_closure` accepts one or two arguments, the input graph G and optionally the option `weighted=true` (or simply `weighted`) or the option `weighted=false` (which is the default). The command returns the transitive closure T of the input graph G. If G is a digraph, then T is also a digraph. When `weighted=true` is specified, the graph T is weighted such that the weight of edge $v\ w \in E'$ is equal to the length (or weight, if G is weighted) of the shortest path from v to w in G.

The lengths/weights of the shortest paths are obtained by the command `allpairs_distance` if G is weighted resp. the command `vertex_distance` if G is unweighted. Therefore T is constructed in at most $O(|V|^3)$ time if `weighted=true` resp. $O(|V|\,|E|)$ time if `weighted=false`.

```
> G:=digraph([1,2,3,4,5,6],%{[1,2],[2,3],[2,4],[4,5],[3,5]%})
```

a directed unweighted graph with 6 vertices and 5 arcs

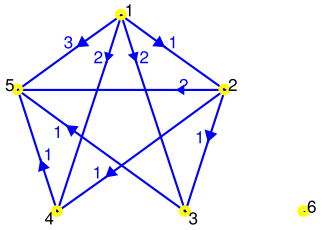
```
> draw_graph(G)
```




```
> T:=transitive_closure(G,weighted)
```

a directed weighted graph with 6 vertices and 9 arcs

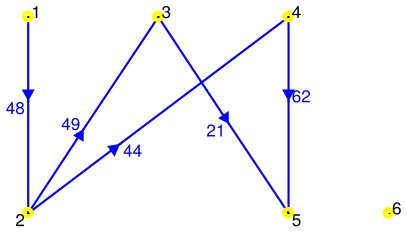
```
> draw_graph(T)
```



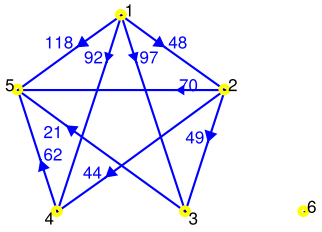

```
> G:=assign_edge_weights(G,1,99)
```

a directed weighted graph with 6 vertices and 5 arcs

```
> draw_graph(G)
```




```
> draw_graph(transitive_closure(G,weighted=true))
```



1.10.10. Line graph

The command `line_graph` is used for constructing the line graph of an undirected graph $G(V, E)$.

`line_graph` accepts the input graph G as its only argument and returns the line graph $L(G)$ with $|E|$ distinct vertices, one vertex for each edge in E. Two vertices v_1 and v_2 in $L(G)$ are adjacent if and only if the corresponding edges $e_1, e_2 \in E$ have a common endpoint. The vertices in $L(G)$ are labeled with strings in form `v-w`, where $e = vw \in E$.

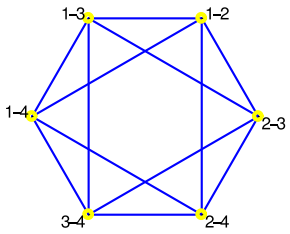
```
> K4:=complete_graph([1,2,3,4])
```

an undirected unweighted graph with 4 vertices and 6 edges

```
> L:=line_graph(K4)
```

an undirected unweighted graph with 6 vertices and 12 edges

```
> draw_graph(L,spring)
```



1.10.11. Plane dual graph

The command `plane_dual` is used for constructing the plane dual graph of an undirected biconnected planar graph.

`plane_dual` accepts one argument, the input graph $G(V, E)$ or the list F of faces of a planar embedding of G, and returns the graph H with faces of G as its vertices in which two vertices are adjacent if and only if they share an edge as faces of G.

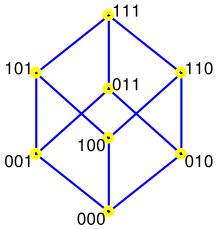
Note that the concept of dual graph is usually defined for multigraphs[1.3]. By the strict definition, every planar multigraph has the corresponding dual multigraph; moreover, the dual of the latter is equal to the former. Since Giac does not support multigraphs, a simplified definition suitable for simple graphs is used.

The algorithm runs in $O(|V|^2)$ time.

```
> H:=hypercube_graph(3)
```

an undirected unweighted graph with 8 vertices and 12 edges

```
> draw_graph(H,spring)
```

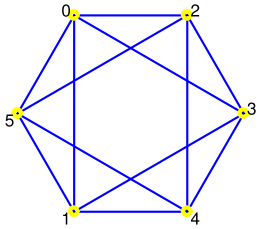


The cube has six faces, hence its plane dual graph D has six vertices. Also, every face obviously shares an edge with exactly four other faces, so the degree of each vertex in D is equal to 4.

```
> D:=plane_dual(H)
```

an undirected unweighted graph with 6 vertices and 12 edges

```
> draw_graph(D,spring)
```



A graph isomorphic to D is obtained when passing a list of faces of H to the `plane_dual` command. The order of vertices is determined by the order of faces.

```
> is_planar(H,F); F
```

$$\text{true},\begin{pmatrix} 010 & 000 & 001 & 011 \\ 001 & 000 & 100 & 101 \\ 010 & 011 & 111 & 110 \\ 100 & 000 & 010 & 110 \\ 111 & 011 & 001 & 101 \\ 101 & 100 & 110 & 111 \end{pmatrix}$$

```
> is_isomorphic(plane_dual(F),D)
```

1.3. See https://en.wikipedia.org/wiki/Dual_graph for the strict definition of plane dual graph.

true

1.11. RANDOM GRAPHS

1.11.1. Random general graphs

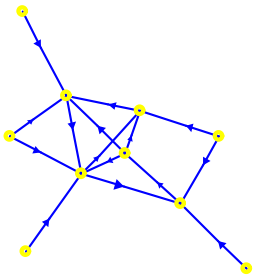
The commands `random_graph` and `random_digraph` are used for generating general graphs at random.

`random_graph` and `random_digraph` both accept two arguments. The first argument is a positive integer n or a list of labels L of length n. The second argument is a positive real number $p<1$ or a positive integer m. The command returns a random (di)graph on n vertices (using elements of L as vertex labels) in which each edge/arc is inserted with probability p or which contains exactly m edges/arcs chosen at random.

```
> G:=random_digraph(10,0.2)
```

a directed unweighted graph with 10 vertices and 15 arcs

```
> draw_graph(G,spring,labels=false)
```



```
> G:=random_graph(1000,0.01)
```

an undirected unweighted graph with 1000 vertices and 4922 edges

```
> is_connected(G)
```

true

```
> is_biconnected(G)
```

false

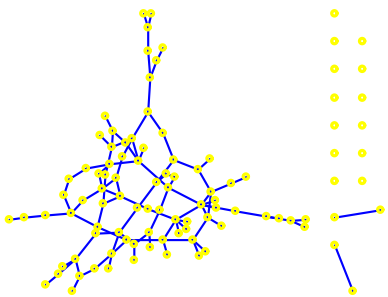
```
> G:=random_graph(100,99)
```

an undirected unweighted graph with 100 vertices and 99 edges

```
> is_tree(G)
```

false

```
> draw_graph(G,spring)
```



1.11.2. Random bipartite graphs

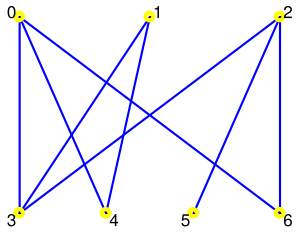
The command `random_bipartite_graph` is used for generating bipartite graphs at random.

`random_bipartite_graph` accepts two arguments. The first argument is either a positive integer n or a list of two positive integers a and b. The second argument is either a positive real number $p<1$ or a positive integer m. The command returns a random bipartite graph on n vertices (or with two partitions of sizes a and b) in which each possible edge is present with probability p (or m edges are inserted at random).

```
> G:=random_bipartite_graph([3,4],0.5)
```

an undirected unweighted graph with 7 vertices and 8 edges

```
> draw_graph(G)
```




```
> G:=random_bipartite_graph(30,60)
```

an undirected unweighted graph with 30 vertices and 60 edges

1.11.3. Random trees

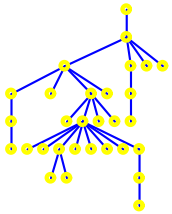
The command `random_tree` is used for generating tree graphs uniformly at random.

`random_tree` accepts a positive integer n as its only argument and returns a random tree generated on n nodes (i.e. inserts $n-1$ edges in the initially empty graph).

```
> T:=random_tree(30)
```

an undirected unweighted graph with 30 vertices and 29 edges

```
> draw_graph(T,labels=false)
```



1.11.4. Random planar graphs

The command `random_planar_graph` is used for generating random planar graphs, controlling edge density and vertex connectivity.

`random_planar_graph` accepts two or three arguments, a positive integer n (or a list L of length n), a positive real number $p<1$ and optionally an integer $k\in\{0,1,2,3\}$ (by default, $k=1$). The command returns a random k-connected planar graph on n vertices (using elements of L as vertex labels).

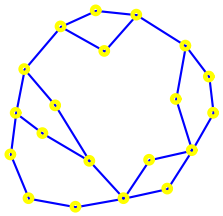
The result is obtained by first generating a random maximal planar graph and then attempting to remove each edge with probability p, maintaining the k-connectivity of the graph (if $k=0$, the graph may be disconnected). The running time is $O(|V|)$ if $k=0$, $O(|V|^2)$ if $k\in\{1,2\}$ and $O(|V|^3)$ if $k=3$.

The following command line generates a biconnected planar graph.

```
> G:=random_planar_graph(20,0.8,2)
```

an undirected unweighted graph with 20 vertices and 25 edges

```
> draw_graph(G,planar,labels=false)
```

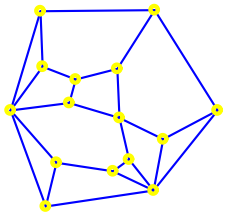


The command line below generates a triconnected planar graph.

```
> G:=random_planar_graph(15,0.9,3)
```

an undirected unweighted graph with 15 vertices and 25 edges

```
> draw_graph(G,planar,labels=false)
```



The next command line generates a disconnected planar graph with high probability.

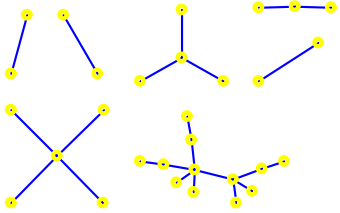
```
> G:=random_planar_graph(30,0.9,0)
```

an undirected unweighted graph with 30 vertices and 23 edges

```
> is_forest(G)
```

true

```
> draw_graph(G,spring,labels=false)
```



By default, a connected planar graph is generated, like in the following example.

```
> G:=random_planar_graph(15,0.618)
```

an undirected unweighted graph with 15 vertices and 19 edges

```
> is_connected(G)
```

true

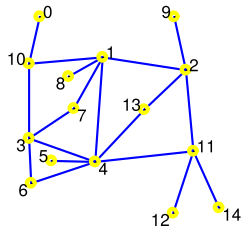
```
> is_biconnected(G)
```

false

```
> articulation_points(G)
```

[1, 2, 4, 10, 11]

```
> draw_graph(G,planar)
```



1.11.5. Random graphs from the given degree sequence

The command `random_sequence_graph` is used for generating a random undirected graph from the given degree sequence.

`random_sequence_graph` accepts the degree sequence (a list of nonnegative integers) as its only argument. It returns an asimptotically uniform random graph with the given degree sequence in almost linear time, using the algorithm developed by Bayati et al. [1].

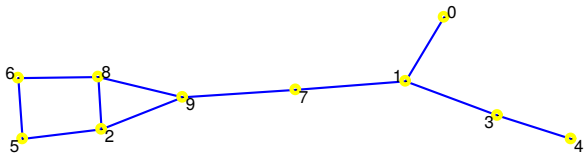
```
> s:=[1,3,3,2,1,2,2,2,3,3]:; is_graphic_sequence(s)
```

Done, true

```
> G:=random_sequence_graph(s)
```

an undirected unweighted graph with 10 vertices and 11 edges

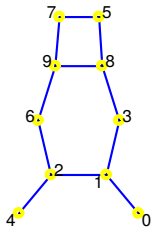
```
> draw_graph(G,spring)
```




```
> H:=random_sequence_graph(s)
```

an undirected unweighted graph with 10 vertices and 11 edges

```
> draw_graph(H,spring)
```



1.11.6. Random regular graphs

The command `random_regular_graph` is used for generating random d-regular graphs on a certain set of vertices, using the algorithm presented in [28][1.4].

1.4. See Algorithm 2 on page 2.

`random_regular_graph` accepts two mandatory arguments, a positive integer n (or a list L of length n) and a nonnegative integer d. Optionally, the option `connected` may be specified as the third argument, indicating that the generated graph must be 1-connected (by default no restriction is made). The command creates n vertices (using elements of L as vertex labels) and returns a random d-regular (connected) graph on these vertices.

Note that a d-regular graph on n vertices exists if and only if $n > d+1$ and $n\,d$ is even. If these conditions are not met, `random_regular_graph` returns an error.

This algorithm generates regular graphs with approximately uniform probability. It means that for $n \to \infty$ and d not growing so quickly with n the probability distribution converges to uniformity.

The runtime of the algorithm is negligible for $n \leqslant 100$. For $n > 200$ the algorithm is considerably slower.

```
> G:=random_regular_graph(16,3)
```

an undirected unweighted graph with 16 vertices and 24 edges

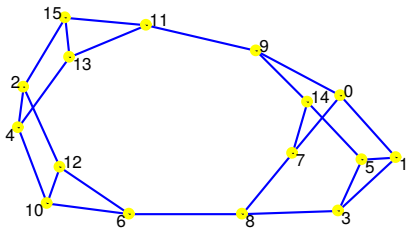
```
> is_regular(G)
```

true

```
> degree_sequence(G)
```

$[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]$

```
> draw_graph(G,spring)
```



1.11.7. Random tournaments

The command `random_tournament` is used for generating random tournaments. A *tournament* of order n is a graph obtained by assigning a direction to each edge in a complete graph K_n (for each edge there are two possible directions).

`random_tournament` accepts a positive integer n (or a list L of length n) as its only argument and returns a random tournament on n vertices. If L is specified, its elements are used to label the vertices.

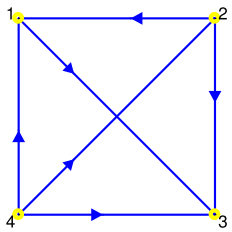
```
> G:=random_tournament([1,2,3,4])
```

a directed unweighted graph with 4 vertices and 6 arcs

```
> is_tournament(G)
```

true

```
> draw_graph(G)
```



1.11.8. Random flow networks

Alternatively, one can combine `random_planar_graph` and `st_ordering` commands to create a random acyclic flow network, embeddedable into plane, with a single source and a single sink.

In the following example, a network on 12 nodes is generated. Firstly, a random planar biconnected graph is generated and embedded into plane.

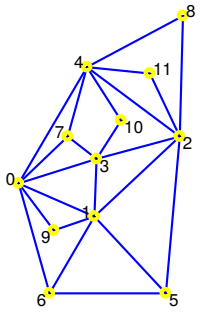
```
> G:=random_planar_graph(12,0.5,2)
```

an undirected unweighted graph with 12 vertices and 23 edges

```
> is_biconnected(G)
```

true

```
> draw_graph(G,planar,pos)
```




```
>  G:=set_vertex_positions(G,pos)
```

an undirected unweighted graph with 12 vertices and 23 edges

Next, two vertices $s, t \in V$ is chosen to be the source and the sink of the network, respectively. These are highlighted and, if $st \notin E$, connected with a temporary edge. Next, the st-ordering (see Section 4.6.3) is computed with respect to s and t. The ordering induces an orientation to G, transforming it to a digraph N. If the temporary edge st was added previously, is is now deleted from N.

```
> s,t:=6,8:;
```

```
> G:=highlight_vertex(G,[s,t])
```

an undirected unweighted graph with 12 vertices and 23 edges

```
> has_edge(G,[s,t])
```

false

```
> G:=add_edge(G,[s,t])
```

an undirected unweighted graph with 12 vertices and 24 edges

```
> st:=st_ordering(G,s,t,N)
```

$[2, 9, 10, 4, 7, 1, 0, 6, 11, 3, 5, 8]$

```
> has_arc(N,[s,t])
```

true

Note that if the previous result was `false`, it would indicate that ts is an arc of N, affecting the order of s and t in the following command line.

```
> N:=delete_arc(N,[s,t])
```

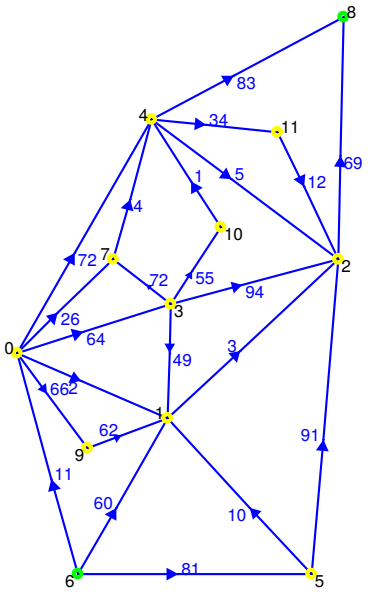
a directed unweighted graph with 12 vertices and 23 arcs

Finally, the arc capacities are set randomly (in this example they are chosen from the set $\{1, 2, ..., 99\} \in \mathbb{Z}$) by applying the `assign_edge_weights` command.

```
> N:=assign_edge_weights(N,1,99)
```

a directed weighted graph with 12 vertices and 23 arcs

```
> draw_graph(N)
```



Because of the st-ordering, it is guaranteed that N contains no directed cycles.

1.11.9. Randomizing edge weights

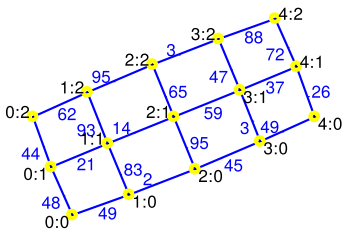
The command `assign_edge_weights` is used for assigning weights to edges of a graph at random.

`assign_edge_weights` accepts two or three arguments. The first argument is always the input graph $G(V, E)$. If only two arguments are given, the second one is an interval `a..b` of real numbers. Otherwise, if three arguments are given, the second resp. the third argument is a positive integer m resp. n. The command operates such that for each edge $e \in E$, its weight is chosen uniformly from the real interval $[a, b)$ or from the set of integers lying between m and n, including both m and n. After assigning weights to all edges, the modified copy of G is returned.

```
> G:=assign_edge_weights(grid_graph(5,3),1,99)
```

an undirected weighted graph with 15 vertices and 22 edges

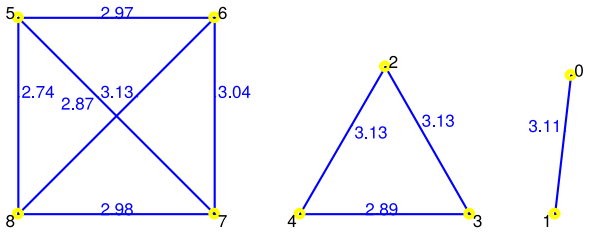
```
> draw_graph(G,spring)
```




```
> G:=assign_edge_weights(graph_complement(complete_graph(2,3,4)),e..pi)
```

an undirected weighted graph with 9 vertices and 10 edges

```
> draw_graph(G)
```


5
2.97
6
2.74
2.87
3.13
3.04
8
2.98
7
2
3.13
3.13
4
2.89
3
0
3.11
1

Chapter 2

Modifying graphs

2.1. Modifying vertices of a graph

2.1.1. Adding and removing single vertices

For adding and removing vertices to/from graphs use the commands `add_vertex` and `delete_vertex`, respectively.

The command `add_vertex` accepts two arguments, a graph $G(V, E)$ and a single label v or a list of labels L, and returns the graph $G'(V \cup \{v\}, E)$ or $G''(V \cup L, E)$ if a list L is given.

```
> K5:=complete_graph([1,2,3,4,5])
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> add_vertex(K5,6)
```

an undirected unweighted graph with 6 vertices and 10 edges

```
> add_vertex(K5,[a,b,c])
```

an undirected unweighted graph with 8 vertices and 10 edges

Note that vertices already present in G won't be added. For example:

```
> add_vertex(K5,[4,5,6])
```

an undirected unweighted graph with 6 vertices and 10 edges

The command `delete_vertex` accepts two arguments, a graph $G(V, E)$ and a single label v or a list of labels L, and returns the graph

$$G'(V \setminus \{v\}, \{e \in E \colon e \text{ is not incident to } v\})$$

or, if L is given,

$$G''(V \setminus L, \{e \in E \colon e \text{ is not incident to any } v \in L\}).$$

If any of the specified vertices does not belong to G, an error is returned.

```
> delete_vertex(K5,2)
```

an undirected unweighted graph with 4 vertices and 6 edges

```
> delete_vertex(K5,[2,3])
```

an undirected unweighted graph with 3 vertices and 3 edges

2.2. Modifying edges of a graph

2.2.1. Adding and removing single edges

For adding and removing edges or arcs to/from graphs use the commands `add_edge` or `add_arc` and `delete_edge` or `delete_arc`, respectively.

The command `add_edge` accepts two arguments, an undirected graph $G(V, E)$ and an edge or a list of edges or a trail of edges (entered as a list of vertices), and returns the copy of G with the specified edges inserted. Edge insertion implies creation of its endpoints if they are not already present.

```
> C4:=cycle_graph(4)
```

C4: an undirected unweighted graph with 4 vertices and 4 edges

```
> add_edge(C4,[1,3])
```

C4: an undirected unweighted graph with 4 vertices and 5 edges

```
> add_edge(C4,[1,3,5,7])
```

C4: an undirected unweighted graph with 6 vertices and 7 edges

The command `add_arc` works similarly to `add_edge` but applies only to directed graphs. Note that the order of endpoints in an arc matters.

```
> add_arc(digraph(trail(a,b,c,d,a)),[[a,c],[b,d]])
```

a directed unweighted graph with 4 vertices and 6 arcs

When adding edge/arc to a weighted graph, its weight should be specified alongside its endpoints, or it will be assumed that it equals to 1.

```
> add_edge(graph(%{[[1,2],5],[[3,4],6]%}),[[2,3],7])
```

an undirected weighted graph with 4 vertices and 3 edges

2.2.2. Accessing and modifying edge weights

The commands `get_edge_weight` and `set_edge_weight` are used to access and modify the weight of an edge/arc in a weighted graph, respectively.

`set_edge_weight` accepts three arguments: a weighted graph $G(V, E)$, edge/arc $e \in E$ and the new weight w, which may be any number. It returns the modified copy of G.

The command `get_edge_weight` accepts two arguments, a weighted graph $G(V, E)$ and an edge or arc $e \in E$. It returns the weight of e.

```
> G:=set_edge_weight(graph(%{[[1,2],4],[[2,3],5]%}),[1,2],6)
```

an undirected weighted graph with 3 vertices and 2 edges

```
> get_edge_weight(G,[1,2])
```

6

2.2.3. Contracting edges

The command `contract_edge` is used for contracting (collapsing) edges in a graph.

`contract_edge` accepts two arguments, a graph $G(V, E)$ and an edge/arc $e = (v, w) \in E$, and merges w and v into a single vertex, deleting the edge e. The resulting vertex inherits the label of v. The command returns the modified graph $G'(V \setminus \{w\}, E')$.

```
> K5:=complete_graph(5)
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> contract_edge(K5,[1,2])
```

an undirected unweighted graph with 4 vertices and 6 edges

To contract a set $\{e_1, e_2, ..., e_k\} \subset E$ of edges in G, none two of which are incident (i.e. when the given set is a matching in G), one can use the `foldl` command. In the following example, the complete graph K_5 is obtained from Petersen graph by edge contraction.

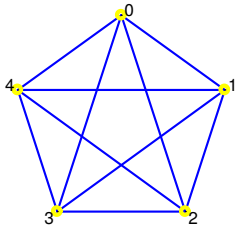
```
> P:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> G:=foldl(contract_edge,P,[0,5],[1,6],[2,7],[3,8],[4,9])
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> draw_graph(G)
```



2.2.4. Subdividing edges

The command `subdivide_edges` is used for subdividing edges of a graph.

`subdivide_edges` accepts two or three arguments, a graph $G(V, E)$, a single edge/arc or a list of edges/arcs in E and optionally a positive integer r (which defaults to 1). Each of the specified edges/arcs will be subdivided with exactly r new vertices, labeled with the smallest available integers. The resulting graph, which is homeomorphic to G, is returned.

If the endpoints of the edge being subdivided have valid coordinates, the coordinates of the inserted vertices will be computed accordingly.

```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> G:=subdivide_edges(G,[2,3])
```

an undirected unweighted graph with 11 vertices and 16 edges

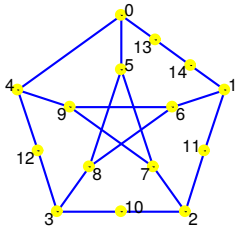
```
> G:=subdivide_edges(G,[[1,2],[3,4]])
```

an undirected unweighted graph with 13 vertices and 18 edges

```
> G:=subdivide_edges(G,[0,1],2)
```

an undirected unweighted graph with 15 vertices and 20 edges

```
> draw_graph(G)
```



2.3. USING ATTRIBUTES

2.3.1. Graph attributes

The graph structure maintains a set of attributes as tag-value pairs which can be accessed and modified by the user.

The command `set_graph_attribute` is used for modifying the existing graph attributes or adding new ones. It accepts two arguments, a graph G and a sequence or list of graph attributes in form `tag=value` where `tag` is any string. Alternatively, attributes may be specified as a sequence of two lists `[tag1,tag2,...]` and `[value1,value2,...]`. The command sets the specified values to the indicated attribute slots, which are meant to represent some global properties of the graph G, and returns the modified copy of G.

The previously set graph attribute values can be fetched with the command `get_graph_attribute` which accepts two arguments: a graph G and a sequence or list of tags. The corresponding values will be returned in a sequence or list, respectively. If some attribute is not set, `undef` is returned as its value.

To list all graph attributes of G for which the values are set, use the `list_graph_attributes` command which takes G as its only argument.

To discard a graph attribute, use the `discard_graph_attribute` command. It accepts two arguments: a graph G and a sequence or list of tags to be cleared, and returns the modified copy of G.

Two tags being used by the CAS commands are directed and weighted, so it is not advisable to overwrite their values using this command (instead, use `make_directed`, `make_weighted` and `underlying_graph` commands). Another attribute used internally is name, which holds the name of the respective graph (as a string).

```
> G:=digraph(trail(1,2,3,1))
```

a directed unweighted graph with 3 vertices and 3 arcs

```
> G:=set_graph_attribute(G,"name"="C3","message"="this is some text")
```

C3: a directed unweighted graph with 3 vertices and 3 arcs

```
> get_graph_attribute(G,"message")
```

this is some text

```
> list_graph_attributes(G)
```

[directed = true, weighted = false, name = $C3$, message = this is some text]

```
> G:=discard_graph_attribute(G,"message")
```

C3: a directed unweighted graph with 3 vertices and 3 arcs

```
> list_graph_attributes(G)
```

[directed = true, weighted = false, name = $C3$]

2.3.2. Vertex attributes

For every vertex of a graph, the list of attributes in form of tag-value pairs is maintained, which can be accessed and modified by the user.

The command `set_vertex_attribute` is used for modifying the existing vertex attributes or adding new ones. It accepts three arguments, a graph $G(V, E)$, a vertex $v \in V$ and a sequence or list of attributes in form `tag=value` where `tag` is any string. Alternatively, attributes may be specified as a sequence of two lists `[tag1,tag2,...]` and `[value1,value2,...]`. The command sets the specified values to the indicated attributes of the vertex v and returns the modified copy of G.

The previously set attribute values for v can be fetched with the command `get_vertex_attribute` which accepts three arguments: G, v and a sequence or list of tags. The corresponding values will be returned in a sequence or list, respectively. If some attribute is not set, `undef` is returned as its value.

To list all attributes of v for which the values are set, use the `list_vertex_attributes` command which takes two arguments, G and v.

The `discard_vertex_attribute` command is used for discarding attribute(s) assigned to some vertex $v \in V$. It accepts three arguments: G, v and a sequence or list of tags to be cleared, and returns the modified copy of G.

The attributes label, color, shape and pos are also used internally. These hold the vertex label, color, shape and coordinates in a drawing, respectively. If the color is not set for a vertex, the latter is drawn in yellow. The shape attribute may have one of the following values: square, triangle, diamond, star or plus. If the shape attribute is not set or has a different value, the circled shape is applied when drawing the vertex.

The following example shows how to change individual labels and colors.

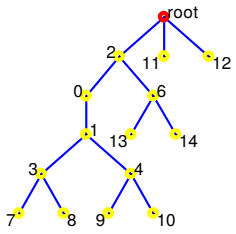
```
> T:=complete_binary_tree(3)
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> T:=set_vertex_attribute(T,5,"label"="root","color"=red)
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> draw_graph(T,tree="root")
```



A vertex may also hold custom attributes.

```
> T:=set_vertex_attribute(T,"root","depth"=3,"shape"="square")
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> list_vertex_attributes(T,"root")
```

$[\text{label} = \text{root}, \text{color} = r\,e\,d, \text{shape} = \text{square}, \text{depth} = 3]$

```
> T:=discard_vertex_attribute(T,"root","color")
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> list_vertex_attributes(T,"root")
```

$[\text{label} = \text{root}, \text{shape} = \text{square}, \text{depth} = 3]$

2.3.3. Edge attributes

For every edge of a graph, the list of attributes in form of key-value pairs is maintained, which can be accessed and modified by the user.

The command `set_edge_attribute` is used for modifying the existing edge attributes or adding new ones. It accepts three arguments, a graph $G(V, E)$, an edge/arc $e \in E$ and a sequence or list of attributes in form `tag=value` where `tag` is any string. Alternatively, attributes may be specified as a sequence of two lists `[tag1,tag2,...]` and `[value1,value2,...]`. The command sets the specified values to the indicated attributes of the edge/arc e and returns the modified copy of G.

The previously set attribute values for e can be fetched with the command `get_edge_attribute` which accepts three arguments: G, e and a sequence or list of tags. The corresponding values will be returned in a sequence or list, respectively. If some attribute is not set, `undef` is returned as its value.

To list all attributes of e for which the values are set, use the `list_edge_attributes` command which takes two arguments, G and e.

To discard attribute(s) assigned to e call the `discard_edge_attribute` command, which accepts three arguments: G, e and a sequence or list of tags to be cleared, and returns the modified copy of G.

The attributes weight, color, style, pos and temp are also used internally. They hold the edge weight, color, line style, the coordinates of the weight label anchor (and also the coordinates of the arrow) and `true` if the edge is temporary. If the color attribute is not set for an edge, the latter is drawn in blue, unless it is a temporary edge, in which case it is drawn in light gray. The style attribute may have one of the following values: dashed, dotted or bold. If the style attribute is not set or has a different value, the solid line style is applied when drawing the edge.

The following example illustrates the possibilities of using edge attributes.

```
> T:=complete_binary_tree(3)
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> T:=set_edge_attribute(T,[1,4],"cost"=12.8,"message"="this is some text")
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> list_edge_attributes(T,[1,4])
```

$[\text{cost} = 12.8, \text{message} = \text{this is some text}]$

```
> T:=discard_edge_attribute(T,[1,4],"message")
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> T:=set_edge_attribute(T,[1,4],"style"="dotted","color"=magenta)
```

an undirected unweighted graph with 15 vertices and 14 edges

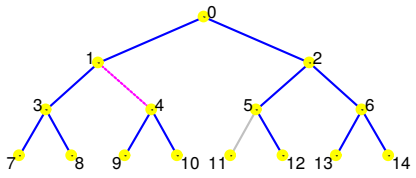
```
> list_edge_attributes(T,[1,4])
```

$[\text{color} = m\,a\,g\,e\,n\,t\,a, \text{style} = \text{dotted}, \text{cost} = 12.8]$

```
> T:=set_edge_attribute(T,[5,11],"temp"=true)
```

an undirected unweighted graph with 15 vertices and 14 edges

```
> draw_graph(T)
```



Chapter 3
Import and export

3.1. Importing graphs

3.1.1. Loading graphs from dot files

The command `import_graph` is used for importing a graph from text file in dot format[3.1].

`import_graph` accepts a string `filename` as its only argument and returns the graph constructed from instructions written in the file `filename` or `undef` on failure. The passed string should contain the path to a file in dot format. The file extension `.dot` may be omitted in the `filename` since dot is the only supported format. The alternative extension is `.gv`,[3.2] which must be explicitly specified.

If a relative path to the file is specified, i.e. if it does not contain a leading forward slash, the current working directory (which can be obtained by calling the `pwd` command) will be used as the reference. The working directory can be changed by using the command `cd`.

For example, assume that the file `example.dot` is saved in the directory `Documents/dot/` with the following contents:

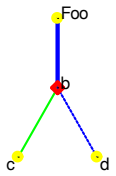
```
graph "Example graph" {
  a [label="Foo"];
  b [shape=diamond,color=red];
  a -- b [style=bold];
  b -- c [color=green];
  b -- d [style=dotted];
}
```

To import the graph, input:

```
> G:=import_graph("Documents/dot/example.dot")
```

Example graph: an undirected unweighted graph with 4 vertices and 3 edges

```
> draw_graph(G)
```



3.1.2. The dot file format overview

Giac has some basic support for dot language[3.3]. Each dot file is used to hold exactly one graph and should consist of a single instance of the following environment:

```
strict? (graph | digraph) name? {
    ...
}
```

3.1. https://en.wikipedia.org/wiki/DOT_(graph_description_language)

3.2. Although it is recommended to use `.gv` as the extension for dot files to avoid a certain confusion between different file types, Giac uses the `.dot` extension because it coincides with the format name. This may change in the future.

3.3. For the complete syntax definition see https://www.graphviz.org/doc/info/lang.html.

The keyword `strict` may be omitted, as well as the `name` of the graph, as indicated by the question marks. The former is used to differentiate between simple graphs (strict) and multigraphs (non-strict). Since Giac supports only simple graphs, `strict` is redundant.

For specifying undirected graphs the keyword `graph` is used, while the `digraph` keyword is used for undirected graphs.

The `graph`/`digraph` environment contains a series of instructions describing how the graph should be built. Each instruction ends with the semicolon (`;`) and has one of the following forms.

syntax	*creates*
`vertex_name [attributes]?`	isolated vertices
`V1 <edgeop> V2 <edgeop> ... <edgeop> Vk [attributes]?`	edges and trails
`graph [attributes]`	graph attributes

Here, `attributes` is a comma-separated list of tag-value pairs in form `tag=value`, `<edgeop>` is `--` for undirected and `->` for directed graphs. Each of `V1`, `V2` etc. is either a vertex name or a set of vertex names in form `{vertex_name1 vertex_name2 ...}`. In the case a set is specified, each vertex from that set is connected to the neighbor operands. Every specified vertex will be created if it does not exist yet.

Any line beginning with `#` is ignored. C-like line and block comments are recognized and skipped as well.

Using the dot syntax it is easy to specify a graph with adjacency lists. For example, the following is the contents of a file which defines the octahedral graph with 6 vertices and 12 edges.

```
# octahedral graph
graph "octahedron" {
  1 -- {3 6 5 4};
  2 -- {3 4 5 6};
  3 -- {5 6};
  4 -- {5 6};
}
```

3.2. EXPORTING GRAPHS

The command `export_graph` is used for saving graphs to disk in dot or LaTeX format.

3.2.1. Saving graphs in dot format

`export_graph` accepts two mandatory arguments, a graph G and a string `filename`, and writes G to the file specified by `filename`, which must be a path to the file, either relative or absolute; in the former case the current working directory will be used as the reference. If only two arguments are given the graph is saved in dot format. The file name may be entered with or without `.dot` extension. The command returns 1 on success and 0 on failure.

```
> export_graph(G,"Documents/dot/copy_of_example")
```

1

3.2.2. Saving graph drawings in LaTeX format

When calling the `export_graph` command, an optional third argument in form `latex[=<params>]` may be given. In that case the drawing of G (obtained by calling the `draw_graph` command) will be saved to the LaTeX file indicated by `filename` (the extension `.tex` may be omitted). Optionally, one can specify a parameter or list of parameters `params` which will be passed to the `draw_graph` command.

For example, let us create a picture of the Sierpiński sieve graph of order $n=5$, i.e. the graph ST_3^5.

```
> G:=sierpinski_graph(5,3,triangle)
```

an undirected unweighted graph with 123 vertices and 243 edges

```
> export_graph(G,"Documents/st53.tex",latex=[spring,labels=false])
```

1

The LaTeX file obtained by exporting a graph is easily converted into an EPS file, which can subsequently be inserted[3.4] in a paper, report or some other document. A Linux user simply needs to launch a terminal emulator, navigate to the directory in which the exported file, in this case `st53.tex`, is stored and enter the following command:

```
latex st53.tex && dvips st53.dvi && ps2eps st53.ps
```

This will produce the (properly cropped) `st53.eps` file in the same directory. Afterwards, it is recommended to enter

```
rm st53.tex st53.aux st53.log st53.dvi st53.ps
```

to delete the intermediate files. The above two commands can be combined in a simple shell script which takes the name of the exported file (without the extension) as its input argument:

```
#!/bin/bash
# convert LaTeX to EPS
latex $1.tex
dvips $1.dvi
ps2eps $1.ps
rm $1.tex $1.aux $1.log $1.dvi $1.ps
```

Assuming that the script is stored under the name `latex2eps` in the same directory as `st53.tex`, to do the conversion it is enough to input:

```
bash latex2eps st53
```

The drawing produced in our example is shown in Figure 3.1.

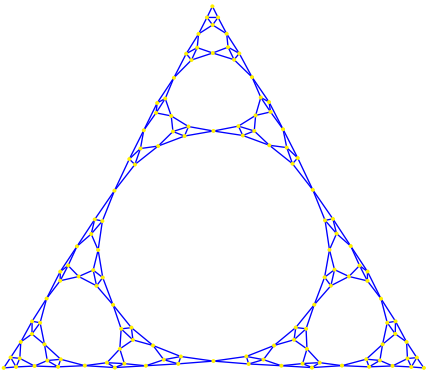


Fig. 3.1. drawing of the Sierpiński graph ST_3^5 using LaTeX and PSTricks

3.4. Alternatively, a PSTricks picture from the body of the `.tex` file can be copied to some other LaTeX document.

Chapter 4

Graph properties

4.1. Basic properties

4.1.1. Listing vertices and edges

The command `vertices` or `graph_vertices` resp. `edges` is used for extracting set of vertices resp. set of edges from a graph. To obtain the number of vertices resp. the number of edges, use the `number_of_vertices` resp. the `number_of_edges` command.

`vertices` or `graph_vertices` accepts a graph $G(V, E)$ as its only argument and returns the set of vertices V in the same order in which they were created.

`edges` accepts one or two arguments, a graph $G(V, E)$ and optionally the identifier `weights`. The command returns the set of edges E (in a non-meaningful order). If `weights` is specified, each edge is paired with the corresponding weight (in this case G must be a weighted graph).

`number_of_vertices` resp. `number_of_edges` accepts the input graph $G(V, E)$ as its only argument and returns $|V|$ resp. $|E|$.

```
> G:=hypercube_graph(2)
```

an undirected unweighted graph with 4 vertices and 4 edges

```
> vertices(G)
```

$[00, 01, 10, 11]$

```
> C:=graph("coxeter")
```

an undirected unweighted graph with 28 vertices and 42 edges

```
> vertices(C)
```

$[a1, a2, a7, z1, a3, z2, a4, z3, a5, z4, a6, z5, z6, z7, b1, b3, b6, b2, b4, b7, b5, c1, c4, c5, c2, c6, c3, c7]$

```
> number_of_vertices(C), number_of_edges(C)
```

$28, 42$

```
> H:=digraph([[0,2.32,0,0.25],[0,0,0,1.32],[0,0.50,0,0],[0.75,0,3.34,0]])
```

a directed weighted graph with 4 vertices and 6 arcs

```
> edges(H)
```

$\{[0, 1], [0, 3], [1, 3], [2, 1], [3, 0], [3, 2]\}$

```
> edges(H,weights)
```

$\{[[0, 1], 2.32], [[0, 3], 0.25], [[1, 3], 1.32], [[2, 1], 0.5], [[3, 0], 0.75], [[3, 2], 3.34]\}$

4.1.2. Vertex degrees

The command `vertex_degree` is used for computing the degree of a vertex, i.e. counting the vertices adjacent to it.

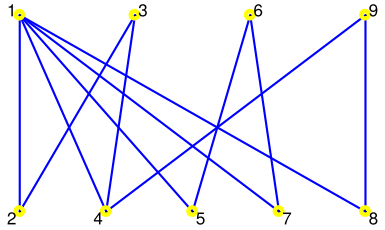
`vertex_degree` accepts two arguments, a graph $G(V,E)$ and a vertex $v \in V$, and returns the cardinality of the set $\{w \in V : (v,w) \in E\}$, i.e. the number of vertices in V which are adjacent to v.

When dealing with directed graphs, one can also use the specialized command `vertex_in_degree` resp. `vertex_out_degree` which accepts the same arguments as `vertex_degree` but returns the number of arcs $(w,v) \in E$ resp. the number of arcs $(v,w) \in E$, where $w \in V$.

```
> G:=graph(trail(1,2,3,4,1,5,6,7,1,8,9,4))
```

an undirected unweighted graph with 9 vertices and 11 edges

```
> draw_graph(G)
```



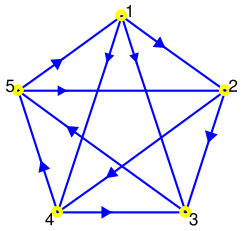

```
> vertex_degree(G,1)
```

5

```
> T:=random_tournament([1,2,3,4,5])
```

a directed unweighted graph with 5 vertices and 10 arcs

```
> draw_graph(T)
```




```
> vertex_out_degree(T,1)
```

3

```
> vertex_in_degree(T,5)
```

2

To obtain the list of degrees of all vertices $v \in V$, use the `degree_sequence` command.

`degree_sequence` accepts a graph $G(V,E)$ as its only argument and returns the list of degrees of vertices from V in the same order as returned by the command `vertices`. If G is a digraph, arc directions are ignored.

```
> degree_sequence(G)
```

$[5,2,2,3,2,2,2,2,2]$

To obtain the maximum degree $\Delta(G)$ or the minimum degree $\delta(G)$ in the graph G, use the command `maximum_degree` or `minimum_degree`, respectively. Both commands accept the input graph G as the only argument.

For example, the command line below shows that Petersen graph is cubic (3-regular).

```
> P:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> minimum_degree(P), maximum_degree(P)
```

3, 3

4.1.3. Regular graphs

The command `is_regular` is used for checking whether a graph is regular, i.e. does each of its vertices have the same number of neighbors.

`is_regular` accepts the input graph $G(V, E)$ as its only argument and returns `true` if $\delta(G) = \Delta(G)$, i.e. if the minimal vertex degree is equal to the maximal vertex degree in G. Else, it returns `false`. The complexity of the algorithm is $O(|V|)$.

```
> is_regular(graph("petersen"))
```

true

```
> is_regular(graph("grotzsch"))
```

false

4.1.4. Vertex adjacency

The command `has_edge` is used for checking if two vertices in an undirected graph are adjacent or not.

`has_edge` accepts two arguments, the input graph $G(V, E)$ and a list `[u,v]` where $u, v \in V$. The command returns `true` if $uv \in E$ and `false` otherwise.

For digraphs, there is the similar command `has_arc` with the same input syntax. Note, however, that the order of vertices u and v matters this time.

```
> G:=graph(trail(1,2,3,4,5,2))
```

an undirected unweighted graph with 5 vertices and 5 edges

```
> has_edge(G,[1,2])
```

true

```
> has_edge(G,[2,1])
```

true

```
> has_edge(G,[1,3])
```

false

```
> D:=digraph(trail(1,2,3,4,5,2))
```

a directed unweighted graph with 5 vertices and 5 arcs

```
> has_arc(D,[1,2])
```

true

```
> has_arc(D,[2,1])
```

false

The command `neighbors` is used for obtaining the list of vertices in a graph that are adjacent to the particular vertex or the complete adjacency structure of the graph, in sparse form.

`neighbors` accepts one or two arguments. The first, mandatory argument is the input graph $G(V, E)$. The second, optional argument is a vertex $v \in V$. The command returns the list of neighbors of v in G if v is given. Otherwise, it returns the list of lists of neighbors for all vertices in V, in order of `vertices(G)`.

Note that `neighbors` works for undirected as well as for directed graphs, but ignores edge directions in the latter case.

```
> neighbors(G,3)
```

$[2, 4]$

```
> neighbors(G)
```

$\{[2], [1, 3, 5], [2, 4], [3, 5], [2, 4]\}$

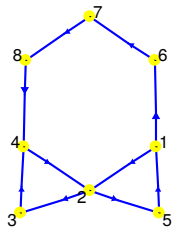
The command `departures` resp. `arrivals` is used for determining all neighbors of a vertex v in a digraph which are the heads resp. the tails of the corresponding arcs.

`departures` resp. `arrivals` accepts one or two arguments, the input digraph $G(V, E)$ and optionally a vertex $v \in V$, and returns the list L_v containing all vertices $w \in V$ for which $vw \in E$ resp. $wv \in E$. If v is omitted, the list of lists L_v for every $v \in V$ is returned.

```
> G:=digraph(trail(1,2,3,4,2,5,1,6,7,8,4))
```

a directed unweighted graph with 8 vertices and 10 arcs

```
> draw_graph(G,spring)
```




```
> departures(G,2); arrivals(G,2); departures(G,1); arrivals(G,1)
```

$[3, 5], [1, 4], [2, 6], [5]$

4.1.5. Edge incidence

The command `incident_edges` is used for obtaining edges incident to the given vertex in a graph.

`incident_edges` accepts two argument, the input graph $G(V, E)$ and a vertex $v \in V$ or a list of vertices $L \subset V$. The command returns the list of edges $e_1, e_2, ..., e_k \in E$ which have v as one of its endpoints.

Note that edge directions are ignored when G is a digraph. To obtain only outgoing or incoming edges, use the commands `departures` and `arrivals`, respectively.

```
> G:=cycle_graph([1,2,3,4,5])
```

C5: an undirected unweighted graph with 5 vertices and 5 edges

```
> incident_edges(G,1)
```

$\{[1, 2], [1, 5]\}$

```
> incident_edges(G,[2,4,5])
```

$\{[1, 2], [1, 5], [2, 3], [3, 4], [4, 5]\}$

```
> G:=random_tournament([1,2,3,4,5])
```

a directed unweighted graph with 5 vertices and 10 arcs

```
> incident_edges(G,2)
```

$$\{[2,1],[2,3],[2,5],[4,2]\}$$

4.2. Algebraic properties

4.2.1. Adjacency matrix

The command `adjacency_matrix` is used for obtaining the adjacency matrix of a graph $G(V,E)$ where $V=\{v_1,v_2,...,v_n\}$.

`adjacency_matrix` accepts the input graph G as its only argument and returns the square matrix $A=[a_{ij}]$ of order n such that, for $i,j=1,2,...,n$,

$$a_{ij}=\begin{cases}1, & \text{if the set } E \text{ contains edge/arc } v_i v_j,\\ 0, & \text{otherwise.}\end{cases}$$

Note that $\mathrm{tr}\,(A)=0$. Also, the adjacency matrix of an undirected graph is always symmetrical.

```
> G:=graph("octahedron")
```

an undirected unweighted graph with 6 vertices and 12 edges

```
> A:=adjacency_matrix(G)
```

$$\begin{pmatrix}0&1&1&1&1&0\\1&0&1&1&0&1\\1&1&0&0&1&1\\1&1&0&0&1&1\\1&0&1&1&0&1\\0&1&1&1&1&0\end{pmatrix}$$

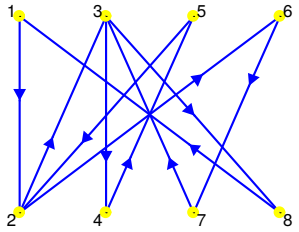
```
> transpose(A)==A
```

true

```
> D:=digraph(trail(1,2,3,4,5,2,6,7,3,8,1))
```

a directed unweighted graph with 8 vertices and 10 arcs

```
> draw_graph(D)
```




```
> A:=adjacency_matrix(D)
```

$$\begin{pmatrix}0&1&0&0&0&0&0&0\\0&0&1&0&0&1&0&0\\0&0&0&1&0&0&0&1\\0&0&0&0&1&0&0&0\\0&1&0&0&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&1&0&0&0&0&0\\1&0&0&0&0&0&0&0\end{pmatrix}$$

```
> transpose(A)==A
```

false

4.2.2. Weight matrix

The command `weight_matrix` is used for obtaining the weight matrix of a weighted graph $G(V, E)$ where $V = \{v_1, v_2, ..., v_n\}$.

`weight_matrix` accepts the input graph G as its only argument and returns the square matrix $M = [m_{ij}]$ of order n such that m_{ij} equals zero if v_i and v_j are not adjacent and the weight of the edge/arc $v_i v_j$ otherwise, for all $i, j = 1, 2, ..., n$ (note that the weight of an edge/arc may be any real number).

Note that $\mathrm{tr}\,(M) = 0$. Also, the weight matrix of an undirected graph is always symmetrical.

```
> G:=graph(%{[[1,2],1],[[2,3],2],[[4,5],3],[[5,2],4]%})
```

an undirected weighted graph with 5 vertices and 4 edges

```
> weight_matrix(G)
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 4 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 4 & 0 & 3 & 0 \end{pmatrix}$$

4.2.3. Incidence matrix

The command `incidence_matrix` is used for obtaining the incidence matrix of a graph.

`incidence_matrix` accepts the input graph $G(V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ and $E = \{e_1, e_2, ..., e_m\}$, as its only argument and returns the $n \times m$ matrix $B = [b_{ij}]$ such that, for all $i = 1, 2, ..., n$ and $j = 1, 2, ..., m$,

$$b_{ij} = \begin{cases} 1, & \text{if the vertex } v_i \text{ is incident to the edge } e_j, \\ 0, & \text{otherwise} \end{cases}$$

if G is undirected resp.

$$b_{ij} = \begin{cases} 1, & \text{if the vertex } v_i \text{ is the head of the arc } e_j, \\ -1, & \text{if the vertex } v_i \text{ is the tail of the arc } e_j, \\ 0, & \text{otherwise} \end{cases}$$

if G is directed.

```
> K4:=complete_graph([1,2,3,4])
```

an undirected unweighted graph with 4 vertices and 6 edges

```
> edges(K4)
```

$$\{[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]\}$$

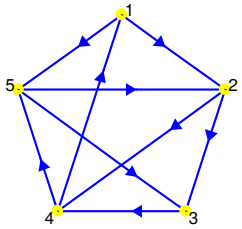
```
> incidence_matrix(K4)
```

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

```
> DG:=digraph(trail(1,2,3,4,5,3),trail(1,5,2,4,1))
```

a directed unweighted graph with 5 vertices and 9 arcs

```
> draw_graph(DG)
```




```
> edges(DG)
```

$$\{[1,2],[1,5],[2,3],[2,4],[3,4],[4,1],[4,5],[5,2],[5,3]\}$$

```
> incidence_matrix(DG)
```

$$\begin{pmatrix} -1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \end{pmatrix}$$

4.2.4. Characteristic polynomial

The command `graph_charpoly` or `charpoly` is used for obtaining the characteristic polynomial of an undirected graph.

`graph_charpoly` or `charpoly` accepts one or two arguments, the input graph $G(V, E)$ and optionally a value or symbol x. The command returns $p(x)$, where p is the characteristic polynomial of the adjacency matrix of G.

```
> G:=graph(%{[1,2],[2,3]%})
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> charpoly(G,x)
```

$$x^3 - 2\,x$$

```
> charpoly(G,3)
```

21

```
> G:=graph("shrikhande")
```

an undirected unweighted graph with 16 vertices and 48 edges

```
> charpoly(G,x)
```

$$x^{16} - 48\,x^{14} - 64\,x^{13} + 768\,x^{12} + 1536\,x^{11} - 5888\,x^{10} - 15360\,x^9 + 23040\,x^8 + 81920\,x^7 - 36864\,x^6 - 245760\,x^5 - 32768\,x^4 + 393216\,x^3 + 196608\,x^2 - 262144\,x - 196608$$

4.2.5. Graph spectrum

The command `graph_spectrum` is used for obtaining the spectrum of eigenvalues of a graph.

`graph_spectrum` accepts the input graph G as its only argument and returns the list in which every element is an eigenvalue of the adjacency matrix of G paired with its multiplicity.

```
> C5:=cycle_graph(5)
```

C5: an undirected unweighted graph with 5 vertices and 5 edges

```
> gs:=graph_spectrum(C5)
```

$$\begin{pmatrix} 2 & 1 \\ \frac{\sqrt{5}-1}{2} & 2 \\ \frac{-\sqrt{5}-1}{2} & 2 \end{pmatrix}$$

```
> p:=charpoly(C5,x)
```

$$x^5 - 5\,x^3 + 5\,x - 2$$

```
> expand(roots(p))==expand(gs)
```

true

The above result indicates that `gs` and `roots(p)` are equal.

4.2.6. Seidel spectrum

The command `seidel_spectrum` is used for obtaining the Seidel spectrum of a graph.

`seidel_spectrum` accepts the input graph $G(V, E)$ as its only argument and returns the list in which every element is an eigenvalue of the matrix $J - I - 2\,A$ paired with its multiplicity. Here J is all-one $n \times n$ matrix, I is the identity matrix of order n, A is the adjacency matrix of G and $n = |V|$.

```
> seidel_spectrum(graph("clebsch"))
```

$$\begin{pmatrix} -3 & 10 \\ 5 & 6 \end{pmatrix}$$

```
> seidel_spectrum(graph("levi"))
```

$$\begin{pmatrix} -5 & 9 \\ -1 & 10 \\ 3 & 9 \\ 5 & 1 \\ 23 & 1 \end{pmatrix}$$

4.2.7. Integer graphs

The command `is_integer_graph` is used for determining if a graph is an integer graph, i.e. if its spectrum consists only of integers.

`is_integer_graph` accepts the input graph G as its only argument. The return value is `true` if G is an integer graph and `false` otherwise.

```
> G:=graph("levi")
```

an undirected unweighted graph with 30 vertices and 45 edges

```
> is_integer_graph(G)
```

true

```
> factor(charpoly(G,x))
```

$$x^{10}\,(x-3)\,(x-2)^9\,(x+2)^9\,(x+3)$$

4.2.8. Equality of graphs

Two graphs are equal if they are both (un)weighted and (un)directed and if the commands `vertices` and `edges` give the same results for both graphs. To test graphs for equality, use the command `graph_equal`.

`graph_equal` accepts two arguments, the input graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, and returns `true` if $G_1 = G_2$ and false otherwise.

```
> G1:=graph([1,2,3],%{[1,2],[2,3]%})
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> G2:=graph([1,3,2],%{[1,2],[2,3]%})
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> graph_equal(G1,G2)
```

false

```
> G3:=graph(trail(1,2,3))
```

an undirected unweighted graph with 3 vertices and 2 edges

```
> graph_equal(G1,G3)
```

true

4.2.9. Graph isomorphism

The command `is_isomorphic` is used for determining whether two graphs are isomorphic.

`is_isomorphic` accepts two or three arguments: the input graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ and optionally an unassigned identifier. The command returns `true` if G_1 and G_2 are isomorphic and `false` otherwise. If the third argument is given and G_1 and G_2 are isomorphic, the list of pairwise vertex matching in G_1 and G_2, representing the isomorphism between the two graphs, is stored to it.

This command, as well as the commands `canonical_labeling` and `graph_automorphisms` described below, is using `nauty` library developed by Brendan McKay [22], which is one of the fastest implementations for graph isomorphism.

For example, entering the command line below one shows that Petersen graph is isomorphic to Kneser graph $K(5, 2)$.

```
> is_isomorphic(graph("petersen"),kneser_graph(5,2))
```

true

In the following example, G_1 and G_3 are isomorphic while G_1 and G_2 are not isomorphic.

```
> G1:=graph(trail(1,2,3,4,5,6,1,3))
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> G2:=graph(trail(1,2,3,4,5,6,1,4))
```

an undirected unweighted graph with 6 vertices and 7 edges

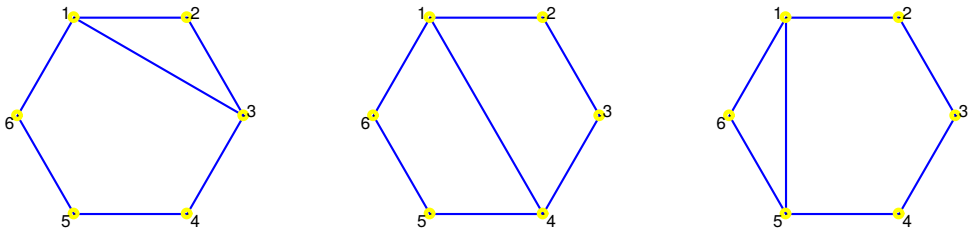
```
> G3:=graph(trail(1,2,3,4,5,6,1,5))
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> draw_graph(G1,circle)
> draw_graph(G2,circle)
```

```
> draw_graph(G3,circle)
```

The drawings are ordered from left to right.




```
> is_isomorphic(G1,G2)
```

false

```
> is_isomorphic(G1,G3)
```

true

```
> is_isomorphic(G1,G3,mapping):; mapping
```

Done, $[1=5, 2=6, 3=1, 4=2, 5=3, 6=4]$

In the next example, D_1 and D_3 are isomorphic while D_1 and D_2 are not isomorphic.

```
> D1:=digraph(trail(1,2,3,1,4,5))
```

a directed unweighted graph with 5 vertices and 5 arcs

```
> D2:=digraph(trail(1,2,3,4,5,3))
```

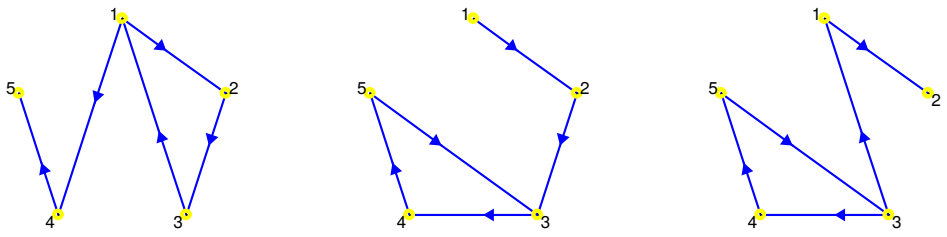
a directed unweighted graph with 5 vertices and 5 arcs

```
> D3:=digraph([1,2,3,4,5],trail(3,4,5,3,1,2))
```

a directed unweighted graph with 5 vertices and 5 arcs

```
> draw_graph(D1,circle)
> draw_graph(D2,circle)
> draw_graph(D3,circle)
```

The drawings are ordered from left to right.




```
> is_isomorphic(D1,D2)
```

false

```
> is_isomorphic(D1,D3)
```

true

Graph isomorphism testing in nauty is based on computing the canonical labelings for the input graphs. These are in fact permutations of the respective sets of vertices which, when applied, turn the input graphs into their canonical representations. The input graphs are isomorphic if and only if their canonical representations share the same edge structure.

The `canonical_labeling` command is used for obtaining the canonical labeling of a graph as a permutation, which can be used with the `isomorphic_copy` command. `canonical_labeling` accepts the input graph as its only argument.

In the next example it is demonstrated how to prove that G_1 and G_3 are isomorphic by comparing their canonical representations H_1 and H_3 with `graph_equal` command. Before testing H_1 and H_3 for equality, their vertices have to be relabeled so that the command `vertices` gives the same result for both graphs.

```
> L1:=canonical_labeling(G1)
```

$$[4, 3, 5, 1, 2, 0]$$

```
> L3:=canonical_labeling(G3)
```

$$[2, 1, 3, 5, 0, 4]$$

```
> H1:=relabel_vertices(isomorphic_copy(G1,L1),[1,2,3,4,5,6])
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> H3:=relabel_vertices(isomorphic_copy(G3,L3),[1,2,3,4,5,6])
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> graph_equal(H1,H3)
```

true

Isomorphism testing with `nauty` is very fast and can be used for large graphs, as in the example below.

```
> G:=random_graph(10000,0.01)
```

an undirected unweighted graph with 10000 vertices and 499867 edges

```
> H:=isomorphic_copy(G,randperm(10000))
```

an undirected unweighted graph with 10000 vertices and 499867 edges

```
> is_isomorphic(G,H)
```

true

1.67 sec

4.2.10. Graph automorphisms

The command `graph_automorphisms` is used for finding generators of the automorphism group of a graph.

`graph_automorphisms` accepts the input graph G as its only argument and returns a list containing the generators of $\mathrm{Aut}(G)$, the automorphism group of G. Each generator is given as a list of cycles, which can be turned to a permutation by calling the `cycles2permu` command.

```
> g:=graph_automorphisms(graph("petersen"))
```

$$\{\begin{pmatrix} 3 & 7 \\ 4 & 5 \\ 8 & 9 \end{pmatrix}, \begin{pmatrix} 2 & 6 \\ 3 & 8 \\ 4 & 5 \\ 7 & 9 \end{pmatrix}, \begin{pmatrix} 1 & 4 \\ 2 & 3 \\ 6 & 9 \\ 7 & 8 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 2 & 4 \\ 5 & 6 \\ 7 & 9 \end{pmatrix}\}$$

```
> cycles2permu(g[2])
```

$$[0, 4, 3, 2, 1, 5, 9, 8, 7, 6]$$

Frucht graph (see the page 22) is an example of a graph with automorphism group containing only the identity, so the set of its generators is empty:

```
> graph_automorphisms(lcf_graph([-5,-2,-4,2,5,-2,2,5,-2,-5,4,2]))
```

{}

4.3. CONNECTIVITY

4.3.1. Vertex connectivity

The commands `is_connected`, `is_biconnected` and `is_triconnected` are used for determining if a graph is connected, biconnected or triconnected, respectively.

All three commands accept only one argument, the input graph $G(V, E)$. They return `true` if G possesses the required type of connectivity and `false` otherwise.

G is *connected* or *1-connected* if for every pair $v, w \in V$ there exists a path with endpoints u and v in G or in the underlying graph of G if the latter is directed.

G is *biconnected* or *2-connected* if it remains connected after removing a vertex from G.

G is *triconnected* or *3-connected* if it remains connected after removing a pair of distinct vertices from G.

The strategy for checking 1- and 2-connectivity is to use depth-first search (see [14] and [29]). Both algorithms run in $O(|V|+|E|)$ time. The algorithm for checking 3-connectivity is quite simple but less efficient: it works by choosing a vertex $v \in V$ and checking if the subgraph induced by $V \setminus \{v\}$ is biconnected, moving on to the next vertex if so, and repeating the process until all vertices are visited exactly once or a non-biconnected subgraph is found for some v. In the latter case the input graph is not triconnected. The complexity of this algorithm is $O(|V|\,|E|)$.

```
> G:=graph_complement(complete_graph(2,3,4))
```

an undirected unweighted graph with 9 vertices and 10 edges

```
> is_connected(G)
```

false

```
> C:=connected_components(G)
```

$\{[0, 1], [2, 3, 4], [5, 6, 7, 8]\}$

```
> H:=induced_subgraph(G,C[2])
```

an undirected unweighted graph with 4 vertices and 6 edges

```
> is_connected(H)
```

true

```
> is_biconnected(path_graph(5))
```

false

```
> is_biconnected(cycle_graph(5))
```

true

```
> is_triconnected(graph("petersen"))
```

true

```
> is_triconnected(cycle_graph(5))
```

false

4.3.2. Connected and biconnected components

The command `connected_components` resp. `biconnected_components` is used for decomposing a graph into connected resp. biconnected components.

`connected_components` resp. `biconnected_components` accept the input graph $G(V, E)$ as its only argument and returns the minimal partition $\{V_1, V_2, ..., V_k\}$ of V such that the subgraph $G_i \subset G$ induced by V_i is connected resp. biconnected for each $i = 1, 2, ..., k$. The partition is returned as a list of lists $V_1, V_2, ..., V_k$.

The connected components of G are easily obtained by depth-first search in $O(|V| + |E|)$ time. To find the biconnected components of G, Tarjan's algorithm is used [29], which also runs in linear time.

```
> G:=graph_complement(complete_graph(3,5,7))
```

an undirected unweighted graph with 15 vertices and 34 edges

```
> is_connected(G)
```

false

```
> C:=connected_components(G)
```

$\{[0, 1, 2], [3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14]\}$

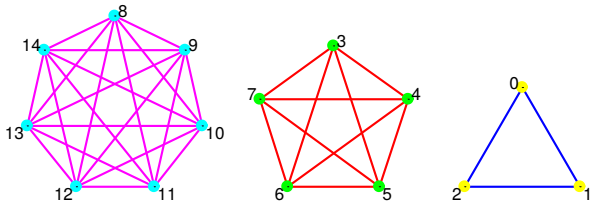
```
> G:=highlight_subgraph(G,induced_subgraph(G,C[1]))
```

an undirected unweighted graph with 15 vertices and 34 edges

```
> G:=highlight_subgraph(G,induced_subgraph(G,C[2]),magenta,cyan)
```

an undirected unweighted graph with 15 vertices and 34 edges

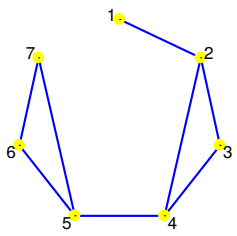
```
> draw_graph(G)
```




```
> H:=graph(trail(1,2,3,4,2),trail(4,5,6,7,5))
```

an undirected unweighted graph with 7 vertices and 8 edges

```
> draw_graph(H)
```




```
> is_biconnected(H)
```

false

```
> biconnected_components(H)
```

$$\{[1,2],[2,3,4],[4,5],[5,6,7]\}$$

4.3.3. Graph rank

The command `graph_rank` is used for computing the rank of a graph.

`graph_rank` accepts one or two arguments, the input graph $G(V,E)$ and optionally a set of edges $S \subset E$ (by default $S = E$), and returns $|V| - k$ where k is the number of connected components of the spanning subgraph of G with edge set S.

```
> G:=graph(%{[1,2],[3,4],[4,5]%})
```

an undirected unweighted graph with 5 vertices and 3 edges

```
> graph_rank(G)
```

3

```
> graph_rank(G,[[1,2],[3,4]])
```

2

4.3.4. Articulation points

The command `articulation_points` is used for obtaining the articulation points of a graph, i.e. cut vertices, if any.

`articulation_points` accepts the input graph $G(V,E)$ as its only argument and returns the list of articulation points of G. A vertex $v \in V$ is an *articulation point* of G if the subgraph $H \subset G$ induced by $V \setminus \{v\}$ is disconnected.

The articulation points of G are found by depth-first search in $O(|V| + |E|)$ time [14].

```
> articulation_points(path_graph([1,2,3,4]))
```

$[2,3]$

```
> articulation_points(cycle_graph(1,2,3,4))
```

[]

4.3.5. Strongly connected components

The command `strongly_connected_components` is used for decomposing a graph into strongly connected components. A (di)graph H is *strongly connected* if for each pair (v,w) of distinct vertices in H there is a (directed) path from v to w in H.

`strongly_connected_components` accepts the input graph $G(V,E)$ as its only argument and returns the minimal partition $\{V_1, V_2, ..., V_k\}$ of V such that the subgraph $G_i \subset G$ induced by V_i is strongly connected for each $i = 1, 2, ..., k$. The result is returned as a list of lists $V_1, V_2, ..., V_k$.

The strategy is to use TARJAN's algorithm for strongly connected components [29], which runs in $O(|V| + |E|)$ time.

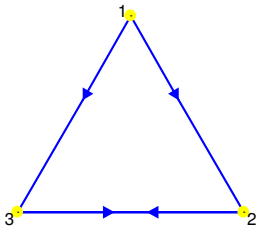
Note that an undirected graph is strongly connected if and only if it is connected.

The command `is_strongly_connected` can be used to determine if the given graph G is strongly connected. It accepts G as its only argument and returns `true` if G has exactly one strongly connected component and `false` otherwise.

```
> G:=digraph([1,2,3],%{[1,2],[1,3],[2,3],[3,2]%})
```

a directed unweighted graph with 3 vertices and 4 arcs

```
> draw_graph(G)
```




```
> is_connected(G)
```

true

```
> is_strongly_connected(G)
```

false

```
> strongly_connected_components(G)
```

$\{[1],[2,3]\}$

4.3.6. Edge connectivity

4.4. Trees

4.4.1. Tree graphs

The command `is_tree` is used for determining if the particular graph is a tree. An undirected graph $G(V,E)$ is a *tree* if $|V|=|E|+1$ and G is connected.

`is_tree` accepts the input graph G as its only argument and returns `true` if G is a tree and `false` otherwise.

The only expensive step in the algorithm is determining whether G is connected. The condition $|V|=|E|+1$ is checked first, hence the algorithm runs in $O(|V|)$ time.

```
> is_tree(complete_binary_tree(3))
```

true

```
> is_tree(cycle_graph(5))
```

false

4.4.2. Forest graphs

The command `is_forest` is used for determining if the particular graph is a *forest*, i.e. if its connected components are all trees.

`is_forest` accepts the input graph G as its only argument and returns `true` if G is a forest and `false` otherwise.

The only expensive step in the algorithm is the decomposition of G to connected components. Therefore the algorithm runs in $O(|V|+|E|)$ time.

```
> L:=[]:; for k from 10 to 30 do L.append(random_tree(k)); od:;
```

Done, Done

```
> G:=disjoint_union(op(L))
```

an undirected unweighted graph with 420 vertices and 399 edges

```
> is_connected(G)
```

false

```
> is_forest(G)
```

true

4.4.3. Height of a tree

The command `tree_height` is used for determining the height of a tree with respect to the specified root node. The *height* of a tree is the length of the longest path in that tree that has the root node as one of its endpoints.

`tree_height` accepts two arguments, the input tree graph $G(V, E)$ and a vertex $r \in V$, which is used as the root node. The command returns the height of G with respect to r.

The strategy is to start a depth-first search from the root node and look for the deepest node. Therefore the algorithm runs in $O(|V|)$ time.

```
> G:=random_tree(1000)
```

an undirected unweighted graph with 1000 vertices and 999 edges

```
> r:=rand(1000)
```

296

```
> tree_height(G,r)
```

20

4.4.4. Lowest common ancestor of a pair of nodes

The command `lowest_common_ancestor` determines the *lowest common ancestor* (LCA) of a pair of nodes in a tree, or for every element of a list of such pairs.

`lowest_common_ancestor` accepts two mandatory arguments, the input tree graph $G(V, E)$ and the root node $r \in V$. There are two possibilities for specifying the nodes to operate on: either the nodes $u, v \in V$ are given as the third and the fourth argument, or a list of pairs of nodes in form `[[u1,v1],[u2,v2],...,[uk,vk]]`, where $u_i, v_i \in V$ and $u_i \neq v_i$ for $i = 1, 2, ..., k$, is given as the third argument. The command returns the LCA of u and v or the list containing LCA of every pair of nodes u_i, v_i for $i = 1, 2, ..., k$.

The strategy is to use TARJAN's offline LCA algorithm [30]. The implementation is simple and uses the disjoint-set (union-find) data structure. It runs in nearly linear time.

In the following example, the algorithm efficiency is tested on a large random tree with 10000 nodes. The lowest common ancestors for the list L containing 100 pairs of vertices, chosen at random, need to be determined.

```
> G:=random_tree(10000)
```

an undirected unweighted graph with 10000 vertices and 9999 edges

3.562 sec

```
> V:=vertices(G):; L:=[]:; for k from 1 to 100 do L.append(rand(2,V)); od:;
```

Done, Done, Done

```
> lowest_common_ancestor(G,0,L):;
```

9.409 sec

4.5. Distance

4.5.1. Vertex distance

The command `vertex_distance` is used for computing the length of the shortest path(s) from the source vertex to some other vertex/vertices of a graph.

`vertex_distance` accepts three arguments, the input graph $G(V, E)$, a vertex $v \in V$ called the *source* and a vertex $w \in V$ called the *target*, or a list $L \subset V$ of target vertices. The command returns the distance between v and w as the number of edges in a shortest path from v to w, or the list of distances if a list of target vertices is given.

The strategy is to use breadth-first search starting from the source vertex. Therefore, the algorithm runs in $O(|V|+|E|)$ time.

```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> vertex_distance(G,1,3)
```

2

```
> vertex_distance(G,1,[3,6,9])
```

$[2, 1, 2]$

4.5.2. All-pairs vertex distance

The command `allpairs_distance` is used for computing the matrix of distances between all pairs of vertices in the given (weighted) graph.

`allpairs_distance` accepts the input graph $G(V, E)$ as its only argument and returns a square matrix $D = [d_{ij}]$ with $n = |V|$ rows and columns such that $d_{ij} = \text{distance}(v_i, v_j)$ for all $i, j = 1, 2, ..., n$, where $v_1, v_2, ..., v_n$ are the elements of V. If $v_i v_j \notin E$, then $d_{ij} = +\infty$. The strategy is to apply the algorithm of Floyd and Warshall [12], which runs in $O(|V|^3)$ time.

Note that, if G is weighted, it must not contain negative cycles.

```
> G:=graph([1,2,3,4,5],%{[1,2],[1,3],[1,4],[1,5],[2,3],[3,4],[4,5],[5,2]%})
```

an undirected unweighted graph with 5 vertices and 8 edges

```
> allpairs_distance(G)
```

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 \\ 1 & 1 & 0 & 1 & 2 \\ 1 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 & 0 \end{pmatrix}$$

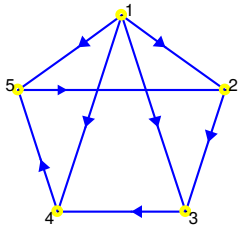
```
> H:=digraph(%{[1,2],[1,3],[1,4],[1,5],[2,3],[3,4],[4,5],[5,2]%})
```

a directed unweighted graph with 5 vertices and 8 arcs

```
> allpairs_distance(H)
```

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ +\infty & 0 & 1 & 2 & 3 \\ +\infty & 3 & 0 & 1 & 2 \\ +\infty & 2 & 3 & 0 & 1 \\ +\infty & 1 & 2 & 3 & 0 \end{pmatrix}$$

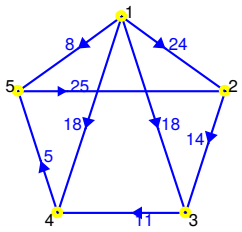
```
> draw_graph(H)
```




```
> H:=assign_edge_weights(H,5,25)
```

a directed weighted graph with 5 vertices and 8 arcs

```
> draw_graph(H)
```




```
> allpairs_distance(H)
```

$$\begin{pmatrix} 0 & 24 & 18 & 18 & 8 \\ +\infty & 0 & 14 & 25 & 30 \\ +\infty & 41 & 0 & 11 & 16 \\ +\infty & 30 & 44 & 0 & 5 \\ +\infty & 25 & 39 & 50 & 0 \end{pmatrix}$$

4.5.3. Diameter of a graph

The command `graph_diameter` is used for determining the maximum distance among all pairs of vertices in a graph.

`graph_diameter` accepts the input graph $G(V, E)$ and returns the number $\max\{\text{distance}(u, v) : u, v \in V\}$. If G is disconnected, $+\infty$ is returned.

This command calls `allpairs_distance` and picks the largest element in the resulting matrix. Hence the complexity of the algorithm is $O(|V|^3)$.

```
> graph_diameter(graph("petersen"))
```

2

```
> graph_diameter(cycle_graph(19))
```

9

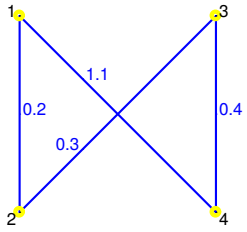
```
> graph_diameter(disjoint_union(graph("petersen"),cycle_graph(19)))
```

$+\infty$

```
> G:=graph(%{[[1,2],0.2],[[2,3],0.3],[[3,4],0.4],[[4,1],1.1]%})
```

an undirected weighted graph with 4 vertices and 4 edges

```
> draw_graph(G)
```




```
> graph_diameter(G)
```

0.9

```
> dijkstra(G,1,4)
```

$[[1, 2, 3, 4], 0.9]$

4.5.4. Girth of a graph

The commands `girth` and `odd_girth` are used for computing the (odd) girth of an undirected unweighted graph.

`girth` resp. `odd_girth` accepts the input graph $G(V, E)$ as its only argument and returns the girth resp. odd girth of G. The (odd) girth of G is defined to be the length of the shortest (odd) cycle in G. If there is no (odd) cycle in G, the command returns $+\infty$.

The strategy is to apply breadth-first search from each vertex of the input graph. The runtime is therefore $O(|V||E|)$.

```
> girth(graph("petersen"))
```

5

```
> G:=hypercube_graph(3)
```

an undirected unweighted graph with 8 vertices and 12 edges

```
> G:=subdivide_edges(G,["000","001"])
```

an undirected unweighted graph with 9 vertices and 13 edges

```
> girth(G)
```

4

```
> odd_girth(G)
```

5

```
> girth(complete_binary_tree(2))
```

$+\infty$

4.6. Acyclic graphs

4.6.1. Checking if a graph is acyclic

The command `is_acyclic` is used for checking that the given digraph has no directed cycle. A directed graph with no directed cycle is said to be *acyclic*.

`is_acyclic` accepts the input digraph $G(V, E)$ as its only argument and returns `true` if G is acyclic and `false` otherwise.

The algorithm attempts to find topological order for its vertices. If that succeeds, the graph is acyclic, otherwise not. The algorithm runs in $O(|V| + |E|)$ time.

```
> is_acyclic(digraph(trail(1,2,3,4,5)))
```

true

```
> is_acyclic(digraph(trail(1,2,3,4,5,2)))
```

false

4.6.2. Topological sorting

The command `topologic_sort` or `topological_sort` is used for finding a linear ordering of vertices of an acyclic digraph which is consistent with the arcs of the digraph.

`topologic_sort` accepts the input graph $G(V, E)$ as its only argument and returns the list of vertices of G in a particular order: a vertex u precedes a vertex v if $uv \in E$, i.e. if there is an arc from u to v.

Note that topological sorting is possible only if the input graph is acyclic. If this condition is not met, `topologic_sort` returns an error. Otherwise, it finds the required ordering by applying KAHN's algorithm [21], which runs in $O(|V| + |E|)$ time.

```
> G:=digraph(%{[c,a],[c,b],[c,d],[a,d],[b,d],[a,b]%})
```

a directed unweighted graph with 4 vertices and 6 arcs

```
> is_acyclic(G)
```

true

```
> topologic_sort(G)
```

$[c, a, b, d]$

4.6.3. st ordering

The command `st_ordering` is used for finding a st-orientation in an undirected biconnected graph with respect to the given source and sink nodes.

`st_ordering` accepts three or four arguments: the input graph $G(V, E)$, a vertex $s \in V$ called the *source*, a vertex $t \in V$ called the *target* or *sink* such that $st \in E$ and optionally an unassigned identifier D. The command returns the permutation σ which defines a particular order of vertices in V. That ordering defines the orientation for each edge $e \in E$, which causes G to become acyclic with a single source s and sink t. The ordering defined by σ is the topological ordering of the resulting digraph. If the optional argument D is given, the digraph is stored to it.

The orientation of $e = uv \in E$ is determined by the ordinals n and m of its endpoints u and v, respectively, which are assigned by the permutation σ. If $n < m$, then u is the head and v is the tail of the corresponding arc, and vice versa otherwise.

Note that the input graph G has a st-orientation if and only if G is biconnected. Furthermore, if the latter is true, a st-orientation can be computed for any pair $s, t \in V$ such that $st \in E$.

If the input graph is not biconnected, `st_ordering` returns an error. Otherwise, it applies the algorithm of EVEN and TARJAN [11], which runs in $O(|V| + |E|)$ time, to find st-ordering for the given pair of vertices.

```
> G:=graph(%{[a,b],[a,c],[a,d],[b,c],[b,d],[c,d]%})
```

an undirected unweighted graph with 4 vertices and 6 edges

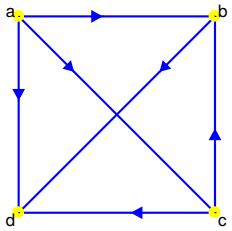
```
> vertices(G)
```

$[a, b, c, d]$

```
> st_ordering(G,a,d,D)
```

$$[0, 2, 1, 3]$$

```
> draw_graph(D)
```



In Section 1.11.8 it is demonstrated how `st_ordering` command can be used for generating random acyclic flow networks.

4.7. Vertex matching

4.7.1. Maximum matching

The command `maximum_matching` is used for finding maximum matching in undirected unweighted graphs.

`maximum_matching` accepts the input graph $G(V, E)$ as its only argument and returns a list of edges $e_1, e_2, ..., e_m \in E$ such that e_i and e_j are not adjacent (i.e. have no common endpoints) for all $1 \leqslant i < j \leqslant m$, under condition that m is maximal. Edges e_k for $k = 1, ..., m$ represent the matched pairs of vertices in G.

The command applies the blossom algorithm[4.1] of Edmonds [10], which finds maximum matching in $O(|V|^2 |E|)$ time.

```
> maximum_matching(graph("octahedron"))
```

$$\begin{pmatrix} 1 & 6 \\ 3 & 2 \\ 5 & 4 \end{pmatrix}$$

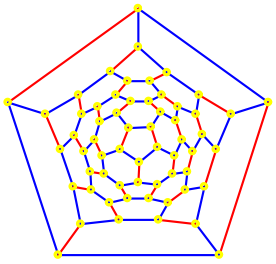
```
> G:=graph("soccerball")
```

an undirected unweighted graph with 60 vertices and 90 edges

```
> M:=maximum_matching(G):; length(M)
```

Done, 30

```
> draw_graph(highlight_edges(G,M),labels=false)
```



```
> G:=random_graph(100,1000)
```

an undirected unweighted graph with 100 vertices and 1000 edges

```
> length(maximum_matching(G))
```

4.1. For a good description of the blossom algorithm, see https://en.wikipedia.org/wiki/Blossom_algorithm.

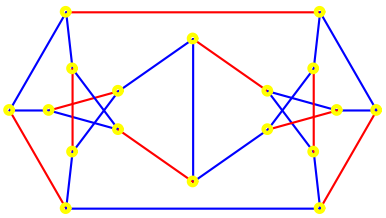
50

13.01 sec

```
> G:=graph("blanusa")
```

an undirected unweighted graph with 18 vertices and 27 edges

```
> draw_graph(highlight_edges(G,maximum_matching(G)),labels=false)
```



4.7.2. Maximum matching in bipartite graphs

The command `bipartite_matching` is used for finding maximum matching in undirected, unweighted bipartite graphs. It applies the algorithm of HOPCROFT and KARP [18], which is more efficient than the blossom algorithm of EDMONDS used by the command `maximum_matching`.

`bipartite_matching` accepts the input graph $G(V, E)$ as its only argument and returns a sequence containing two elements: the size of the maximum matching and the list of edges connecting matched pairs of vertices. The algorithm runs in $O\left(\sqrt{|V|}\,|E|\right)$ time.

```
> G:=graph("desargues")
```

an undirected unweighted graph with 20 vertices and 30 edges

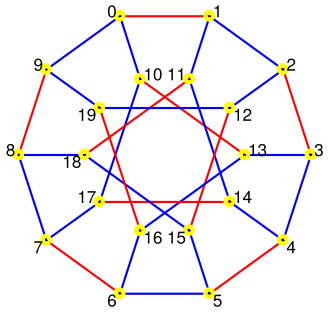
```
> is_bipartite(G)
```

true

```
> n,M:=bipartite_matching(G)
```

$$10, \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \\ 10 & 13 \\ 11 & 18 \\ 12 & 15 \\ 14 & 17 \\ 16 & 19 \end{pmatrix}$$

```
> draw_graph(highlight_edges(G,M))
```



4.8. Cliques

4.8.1. Clique graphs

The graph is a *clique* if it is complete, i.e. if each two of its vertices are adjacent to each other. To check if a graph is a clique one can use the `is_clique` command.

`is_clique` accepts a graph $G(V, E)$ as its only argument and returns `true` if G is a complete graph; else the returned value is `false`.

```
> K5:=complete_graph(5)
```

an undirected unweighted graph with 5 vertices and 10 edges

```
> is_clique(K5)
```

true

```
> G:=delete_edge(K5,[1,2])
```

an undirected unweighted graph with 5 vertices and 9 edges

```
> is_clique(G)
```

false

4.8.2. Triangle-free graphs

The command `is_triangle_free` is used for determining is the particular graph triangle-free. A graph is *triangle-free* if it contains no clique of cardinality equal to 3, and hence no cliques with cardinality greater than two.

`triangle_free` accepts the input graph G as its only argument and returns `true` if G is triangle-free and `false` otherwise.

The strategy is to compute the trace of A^3, where A is the adjacency matrix of G (encoded as a sparse matrix). If $\mathrm{tr}\,(A^3) = 0$, the graph is triangle-free[4.2]. This method is very fast as in practice only one matrix multiplication needs to be carried out completely.

```
> is_triangle_free(graph("soccerball"))
```

true

```
> is_triangle_free(graph("tetrahedron"))
```

false

4.8.3. Maximal cliques

Each subgraph of a graph $G(V, E)$ which is itself a complete graph is called a clique in G. A clique is *maximal* if it cannot be extended by adding more vertices from V to it. To count all maximal cliques in a graph one can use the `clique_stats` command.

`clique_stats` accepts G as the only mandatory argument. If it is the only argument given, the command returns a list of pairs, each pair consisting of two integers: clique cardinality k (first) and the number $n_k > 0$ of k-cliques in G (second). Therefore, the sum of second members of all returned pairs is equal to the total count of all maximal cliques in G. As an optional second argument one may give a positive integer k or an interval m .. n with integer bounds. In the first case only the number of k-cliques for the given k is returned; in the second case, only cliques with cardinality between m and n (inclusive) are counted.

4.2. See https://en.wikipedia.org/wiki/Triangle-free_graph#Triangle_finding_problem

The strategy used to find all maximal cliques is a variant of the algorithm of BRON and KERBOSCH developed by TOMITA et al. [31]. Its worst-case running time is $O(3^{|V|/3})$. However, the performance usually takes only a moment for graphs with 100 vertices or less.

```
> G:=random_graph(50,0.5)
```

an undirected unweighted graph with 50 vertices and 588 edges

```
> clique_stats(G)
```

$$\begin{pmatrix} 3 & 14 \\ 4 & 185 \\ 5 & 370 \\ 6 & 201 \\ 7 & 47 \\ 8 & 5 \end{pmatrix}$$

```
> G:=random_graph(100,0.5)
```

an undirected unweighted graph with 100 vertices and 2461 edges

```
> clique_stats(G,5)
```

3124

```
> G:=random_graph(500,0.25)
```

an undirected unweighted graph with 500 vertices and 31257 edges

```
> clique_stats(G,5..7)
```

$$\begin{pmatrix} 5 & 153444 \\ 6 & 18486 \\ 7 & 355 \end{pmatrix}$$

1.218 sec

4.8.4. Maximum clique

The largest maximal clique in the graph $G(V, E)$ is called *maximum clique*. The command `maximum_clique` can be used to find one in the given graph.

`maximum_clique` accepts the graph G as its only argument and returns maximum clique in G as a list of vertices. The clique may subsequently be extracted from G using the command `induced_subgraph`.

The strategy used to find maximum clique is an improved variant of the classical algorithm by CARRAGHAN and PARDALOS developed by ÖSTERGÅRD [24].

In the following examples, the maximum cliques were obtained almost instantly.

```
> G:=sierpinski_graph(5,5)
```

an undirected unweighted graph with 3125 vertices and 7810 edges

```
> maximum_clique(G)
```

$[1560, 1561, 1562, 1563, 1564]$

```
> G:=random_graph(300,0.3)
```

an undirected unweighted graph with 300 vertices and 13380 edges

```
> maximum_clique(G)
```

$[46, 64, 144, 183, 208, 241, 244, 261]$

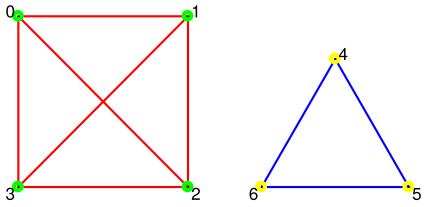
```
> G:=graph_complement(complete_graph(4,3))
```

an undirected unweighted graph with 7 vertices and 9 edges

```
> cliq:=maximum_clique(G)
```

$[0, 1, 2, 3]$

```
> draw_graph(highlight_subgraph(G,induced_subgraph(G,cliq)))
```



4.8.5. Minimum clique covering

The *minimum clique covering* for the graph $G(V, E)$ is the smallest set $S = \{C_1, C_2, ..., C_k\}$ of cliques in G such that for every $v \in V$ there exists $i \leqslant k$ such that $v \in C_i$. Such covering can be obtained by calling the `clique_cover` command.

`clique_cover` accepts graph G as its mandatory argument and returns the smallest possible covering. Optionally, a positive integer may be passed as the second argument. In that case the requirement that k is less or equal to the given integer is set. If no such covering is found, `clique_cover` returns empty list.

The strategy is to find the minimal vertex coloring in the complement G^c of G (note that these two graphs share the same set of vertices). Each set of equally colored vertices in G^c corresponds to a clique in G. Therefore, the color classes of G^c map to the elements $C_1, ..., C_k$ of the minimal clique cover in G.

There is a special case in which G is triangle-free, which is treated separately. Such a graph G contains only 1- and 2-cliques; in fact, every clique covering in G consists of a matching M together with the singleton cliques (i.e. the isolated vertices which remain unmatched). The total number of cliques in the covering is equal to $|V| - |M|$, hence to find the minimal cover one just needs to find maximum matching in G, which can be done in polynomial time.

```
> G:=random_graph(30,0.2)
```

an undirected unweighted graph with 30 vertices and 89 edges

```
> clique_cover(G)
```

{[0, 21], [1, 17], [2, 25, 28], [3, 7, 10], [4, 8], [5, 11, 20], [6, 13, 14], [9, 16, 23], [12, 15, 19], [18, 22], [24, 26], [27, 29]}

To find minimal clique covering in the truncated icosahedral graph it suffices to find maximum matching, since it is triangle-free.

```
> clique_cover(graph("octahedron"))
```

$$\begin{pmatrix} 1 & 3 & 6 \\ 2 & 4 & 5 \end{pmatrix}$$

The vertices of Petersen graph can be covered with five, but not with three cliques.

```
> clique_cover(graph("petersen"),3)
```

[]

```
> clique_cover(graph("petersen"),5)
```

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 9 \\ 5 & 7 \\ 6 & 8 \end{pmatrix}$$

4.8.6. Clique covering number

The command `clique_cover_number` is used for computing the clique covering number of a graph.

`clique_cover_number` accepts the input graph $G(V, E)$ as its only argument and returns the minimum number of cliques in G needed to cover the vertex set V. (More precisely, it calls the `clique_cover` command and returns the length of the resulting list.) This number, denoted by $\theta(G)$, is equal to the chromatic number $\chi(G^c)$ of the complement graph G^c of G.

```
> clique_cover_number(graph("petersen"))
```

5

```
> clique_cover_number(graph("soccerball"))
```

30

```
> clique_cover_number(random_graph(40,0.618))
```

7

4.9. VERTEX COLORING

To *color* vertices of a graph $G(V, E)$ means to assign to each vertex $v \in V$ a positive integer. Each integer represents a distinct color. The key property of a graph coloring is that the colors of adjacent vertices must differ from one another. Two different colorings of G may use different number of colors.

4.9.1. Greedy coloring

The command `greedy_color` is used for coloring vertices of a graph in a greedy fashion.

`greedy_color` accepts one mandatory argument, the input graph G. Optionally, a permutation p of order $|V|$ may be passed as the second argument. Vertices are colored one by one in the order specified by p (or in the default order if p is not given) such that each vertex gets the smallest available color. The list of vertex colors is returned in the order of `vertices(G)`.

Generally, different choices of permutation p produce different colorings. The total number of different colors may not be the same each time. The complexity of the algorithm is $O(|V| + |E|)$.

```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

```
> greedy_color(G)
```

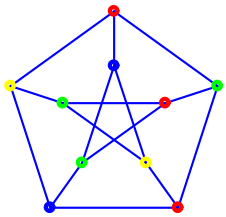
$[1, 2, 1, 2, 3, 2, 1, 3, 3, 2]$

```
> L:=greedy_color(G,randperm(10))
```

$[1, 2, 1, 4, 3, 4, 1, 3, 2, 2]$

Observe that a different number of colors is obtained by executing the last command line. To display the colored graph, input:

```
> draw_graph(highlight_vertex(G,vertices(G),L),labels=false)
```



The first six positive integers are always mapped to the standard Xcas colors, as indicated in Table 4.1. Note that the color 0 (black) and color 7 (white) are swapped; a vertex with color 0 is *uncolored* or *white*, and vertex with color 7 is black. Also note that Xcas will map the numbers greater than 7 to colors too, but the number of available colors is limited.

4.9.2. Minimal coloring

The vertex coloring of G is *minimal* (or *optimal*) if the smallest possible number of colors is used. To obtain such a coloring use the command `minimal_vertex_coloring`.

`minimal_vertex_coloring` accepts one mandatory argument, the graph G. Optionally, a symbol `sto` may be passed as the second argument. The command returns the vertex colors in order of `vertices(G)` or, if the second argument is given, stores the colors as vertex attributes and returns the modified copy of G.

Giac requires the GLPK library[4.3] to solve the minimal vertex coloring problem (MVCP), which is converted to the equivalent integer linear programming problem and solved by using the branch-and-bound method with specific branch/backtrack techniques [8]. Lower and upper bounds for the number of colors n are obtained by finding a maximal clique (n cannot be lower than its cardinality) and by using the heuristic proposed by Brélaz in [3] (which will use at least n colors), respectively. Note that the algorithm performs some randomization when applying heuristics, so coloring a graph several times will not take the same amount of computation time in each instance, generally.

In the following example, the Grotzsch graph is colored with minimal number of colors by first finding the coloring and then assigning it to the graph by using the `highlight_vertex` command.

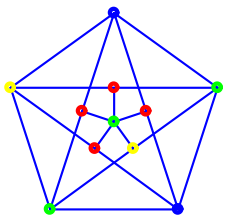
```
> G:=graph("grotzsch")
```

an undirected unweighted graph with 11 vertices and 20 edges

```
> coloring:=minimal_vertex_coloring(G)
```

$$[4, 2, 3, 1, 1, 4, 1, 3, 2, 1, 2]$$

```
> draw_graph(highlight_vertex(G,vertices(G),coloring),labels=false)
```



Solving MVCP for different graphs of exactly the same size (but which do not share the same edge structure) may take quite different time in each instance. Also note that, since vertex coloring problem is NP hard, the algorithm will take exponential time for some graphs.

4.3. GNU Linear Programming Kit, https://www.gnu.org/software/glpk/

value	*color*
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	black

Table 4.1. interpretation of abstract vertex/edge colors in Xcas

4.9.3. Chromatic number

The command `chromatic_number` is used for exact computation and approximation of the chromatic number of a graph.

`chromatic_number` accepts one mandatory argument, the input graph $G(V, E)$, and optionally a second argument. To obtain only upper and lower bound for the chromatic number (which is much faster than computing exactly) the option `approx` or `interval` should be passed as the second argument. Alternatively, an unassigned identifier is passed as the second argument; in that case the corresponding coloring will be stored to it in form of a list of colors of the individual vertices, ordered as in `vertices(G)`.

The command returns the chromatic number χ_G of the graph G in the case of exact computation. If the option `approx` or `interval` is given, an interval `lb..ub` is returned, where `lb` is the best lower bound and `ub` the best upper bound for χ_G found by the algorithm.

The strategy is call `minimal_vertex_coloring` in the case of exact computation. When approximating the chromatic number, the algorithm will establish the lower bound by searching for maximum clique. The timeout for this operation is set to 5 seconds as it can be time consuming. If the maximum clique is not found after that time, the largest clique found is used. Then, an upper bound is established by by using the heuristic proposed by BRÉLAZ in [3]. Obtaining the bounds for χ_G is usually very fast, however the difference between them grows with $|V|$.

Unless the input graph is sparse enough, the algorithm slows down considerably for, say, $|V| > 40$.

```
> chromatic_number(graph("grotzsch"),cols)
```

4

```
> cols
```

$[4, 2, 3, 1, 1, 4, 1, 3, 2, 1, 2]$

```
> G:=random_graph(30,0.75)
```

an undirected unweighted graph with 30 vertices and 313 edges

```
> chromatic_number(G)
```

10

```
> G:=random_graph(300,0.05)
```

an undirected unweighted graph with 300 vertices and 2196 edges

```
> chromatic_number(G,approx)
```

4..7

4.9.4. *k*-coloring

The command `is_vertex_colorable` is used for determining whether the vertices of a graph can be colored with at most k colors.

`is_vertex_colorable` accepts two or three arguments: the input graph $G(V,E)$, a positive integer k and optionally an unassigned identifier. The command returns `true` if G can be colored using at most k colors and `false` otherwise. If an identifier is given, a coloring using at most k colors is stored to it as a list of vertex colors, in the order of `vertices(G)`.

The strategy is to first apply a simple greedy coloring procedure which runs in linear time. If the number of required colors is greater than k, the heuristic proposed by Brélaz in [3] is used, which runs in quadratic time. If the number of required colors is still larger than k, the algorithm attempts to find the chromatic number χ_G using k as the upper bound in the process.

```
> G:=graph("grotzsch")
```

an undirected unweighted graph with 11 vertices and 20 edges

```
> is_vertex_colorable(G,3)
```

false

```
> is_vertex_colorable(G,4)
```

true

```
> G:=random_graph(70,0.2)
```

an undirected unweighted graph with 70 vertices and 469 edges

```
> chromatic_number(G,approx)
```

5..6

```
> is_vertex_colorable(G,5)
```

false

818 msec

From the results of the last two command lines it follows that $\chi_G = 6$. Finding χ_G by utilizing the next command line is simpler, but requires much more time.

```
> chromatic_number(G)
```

6

92.7 sec

4.10. Edge coloring

4.10.1. Minimal coloring

The command `minimal_edge_coloring` is used for finding a minimal edge coloring of edges in a graph, satisfying two conditions: two edges necessary have distinct colors if they are incident to each other and the total number n of colors is minimal. The theorem of Vizing (see [9], Theorem 5.3.2, page 103) implies that every simple undirected graph falls into one of two categories: *1* if $n = \Delta$ or *2* if $n = \Delta + 1$, where Δ is the maximum degree of the graph.

`minimal_edge_coloring` accepts one or two arguments, the input graph $G(V,E)$ and optionally the keyword `sto`. If the latter is given, minimal coloring is stored to the input graph (each edge $e \in E$ gets a color c_e stored as an attribute) and a modified copy of G is returned. Else, the command returns a sequence of two objects: integer `1` or `2`, indicating the category, and the list of edge colors $c_{e_1}, c_{e_2}, ..., c_{e_m}$ according the order of edges $e_1, e_2, ..., e_m \in E$ as returned by the command `edges`.

The strategy is to find minimal vertex coloring of the line graph of G by using the algorithm described in Section 4.9.2.

```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

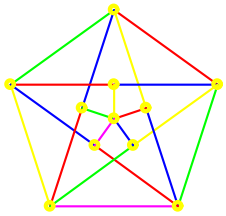
```
> minimal_edge_coloring(G)
```

$2, [1, 2, 3, 2, 3, 3, 4, 1, 2, 3, 1, 4, 1, 4, 2]$

```
> H:=minimal_edge_coloring(graph("grotzsch"),sto)
```

an undirected unweighted graph with 11 vertices and 20 edges

```
> draw_graph(H,labels=false)
```



```
> G:=random_graph(100,0.1)
```

an undirected unweighted graph with 100 vertices and 499 edges

```
> minimal_edge_coloring(G):;
```

20.24 sec

4.10.2. Chromatic index

The command `chromatic_index` is used for computing the chromatic index of an undirected graph, i.e. the minimal number of colors needed to color each edge of the graph such that two incident edges never share the same color.

`chromatic_index` accepts one or two arguments, the input graph $G(E, V)$ and optionally an unassigned identifier. The command returns the chromatic index $\chi'(G)$ of G. If the second argument is given, it specifies the destination for storing minimal edge coloring in form of a list of colors according to the order of edges in E as returned by the command `edges`.

The example below demonstrates how to color the edges of a graph with colors obtained by passing unassigned identifier `c` to `chromatic_index` as the second argument.

```
> G:=graph("petersen")
```

an undirected unweighted graph with 10 vertices and 15 edges

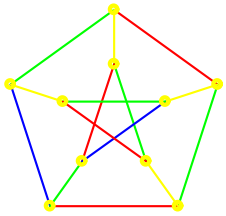
```
> chromatic_index(G)
```

4

```
> chromatic_index(G,c):; E:=edges(G):;
```

Done, Done

```
> for k from 0 to 14 do G:=set_edge_attribute(G,E[k],"color"=c[k]) od:;
> draw_graph(G,labels=false)
```

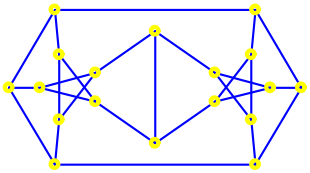


Blanuša snarks, the two graphs with 18 vertices found in 1946 by Danilo Blanuša, were the second and third snarks[4.4] discovered [2]. For almost fifty years, Petersen graph was the only known snark. The second Blanuša snark is available in Giac by passing the string `"blanusa"` to the `graph` command.

```
> G:=graph("blanusa")
```

an undirected unweighted graph with 18 vertices and 27 edges

```
> draw_graph(G,labels=false)
```



```
> chromatic_index(G)
```

4

4.4. A *snark* is connected, bridgeless, cubic graph with chromatic index equal to 4 and usually restricted to have girth at least 5.

Chapter 5

Traversing graphs

5.1. Walks and tours

5.1.1. Eulerian graphs

The command `is_eulerian` is used for determining whether the given graph is an Eulerian graph and for finding Eulerian trails in such graphs.

`is_eulerian` accepts one or two arguments, the input graph $G(V, E)$ and optionally an unassigned identifier T, and returns `true` if G is Eulerian and `false` otherwise. If T is given, the corresponding Eulerian trail is stored to it.

The graph G is Eulerian if it has a trail covering all its edges. Such a trail is called *Eulerian trail*. An Eulerian trail may be closed, in which case it is called *Eulerian cycle* or *circuit*. Note that every edge $e \in E$ must be visited, i.e. "strolled through", exactly once. The edge endpoints (i.e. the vertices in G) may, however, be visited more than once.

The strategy is to apply Hierholzer's algorithm for finding an Eulerian path [16]. It works by covering one cycle at a time in the input graph. The required time is $O(|E|)$.

```
> is_eulerian(complete_graph(4))
```

false

```
> purge(T):; is_eulerian(complete_graph([1,2,3,4,5]),T); T
```

Done, true, $[1, 2, 3, 4, 1, 5, 2, 4, 5, 3, 1]$

5.1.2. Hamiltonian graphs

The command `is_hamiltonian` is used for checking hamiltonicity of an undirected graph. The command can also construct a Hamiltonian cycle in the input graph if the latter is Hamiltonian.

`is_hamiltonian` accepts one or two arguments, the input graph $G(V, E)$ and optionally an unassigned identifier. The command returns `true` if G is Hamiltonian and `false` otherwise. When failing to determine whether G is Hamiltonian or not, `is_hamiltonian` returns `undef`. If an identifier is passed as the second argument, a Hamiltonian cycle is stored to it.

The strategy is to apply some (non)hamiltonicity criteria presented in DeLeon [7] before resorting to the definitive but NP-hard algorithm. If G is not biconnected, it is not Hamiltonian. Else, the criterion of Dirac is applied: if $\delta(G) \geqslant \frac{|V|}{2}$, where $\delta(G) = \min\{\deg(v) : v \in V\}$, then G is Hamiltonian. Else, if G is bipartite with vertex partition $V = V_1 \cup V_2$ and $|V_1| \neq |V_2|$, then G is not Hamiltonian. Else, the criterion of Ore is applied: if $\deg(u) + \deg(v) \geqslant n$ holds for every pair u, v of non-adjacent vertices from V, then G is Hamiltonian. Else, the theorem of Bondy and Chvátal is applied: if the closure $\mathrm{cl}(G)$ of G (obtained by finding a pair u, v of non-adjacent vertices from V such that $\deg(u) + \deg(v) \geqslant n$, adding a new edge uv to E and repeating the process until exhaustion) is Hamiltonian, then G is Hamiltonian. (Note that in this case the previously tried criteria are applied to $\mathrm{cl}(G)$; since the vertex degrees in $\mathrm{cl}(G)$ are generally higher than those in G, the probability of success also rises.) Else, if the edge density of G is large enough, the criterion of Nash and Williams is applied: if $\delta(G) \geqslant \max\left\{\frac{n+2}{3}, \beta\right\}$, where β is the independence number of G, then G is Hamiltonian. If all of the above criteria fail, the brute force algorithm is applied; essentially, the command `traveling_salesman` is called to find a Hamiltonian cycle or to determine that none exist.

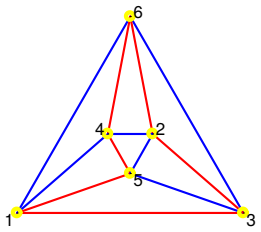
```
> is_hamiltonian(graph("soccerball"))
```

true

```
> is_hamiltonian(graph("octahedron"),hc)
```

true

```
> draw_graph(highlight_trail(graph("octahedron"),hc))
```




```
> is_hamiltonian(graph("herschel"))
```

false

```
> is_hamiltonian(graph("petersen"))
```

false

```
> is_hamiltonian(hypercube_graph(9))
```

true

6.04 sec

5.2. Optimal routing

5.2.1. Shortest paths in unweighted graphs

The command `shortest_path` is used to find the shortest path between two vertices in an undirected unweighted graph.

`shortest_path` accepts three arguments: a graph $G(V, E)$, the source vertex $s \in V$ and the target vertex $t \in V$ or a list T of target vertices. The shortest path from source to target is returned. If more targets are specified, the list of shortest paths from the source to each of these vertices is returned.

The strategy is to run breadth-first traversal on the graph G starting from the source vertex s. The complexity of the algorithm is therefore $O(|V| + |E|)$.

```
> G:=graph("dodecahedron")
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> shortest_path(G,1,16)
```

$[1, 6, 11, 16]$

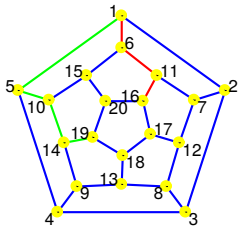
```
> paths:=shortest_path(G,1,[16,19])
```

$\{[1, 6, 11, 16], [1, 5, 10, 14, 19]\}$

```
> H:=highlight_trail(G,paths,[red,green])
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> draw_graph(H)
```



5.2.2. Cheapest paths in weighted graphs

The command `dijkstra` is used for finding the cheapest path between two distinct vertices of an undirected weighted graph.

`dijkstra` accepts two or three arguments: a weighted graph $G(V,E)$ with nonnegative weights, a vertex $s \in V$ and optionally a vertex $t \in V$ or list T of vertices in V. It returns the cheapest path from s to t or, if more target vertices are given, the list of such paths to each target vertex $t \in T$, computed by Dijkstra's algorithm in $O(|V|^2)$ time. If no target vertex is specified, all vertices in $V \setminus \{s\}$ are assumed to be targets.

A cheapest path from s to t is represented with a list `[[v1,v2,...,vk],c]` where the first element consists of path vertices with $v_1 = s$ and $v_k = t$, while the second element c is the weight (cost) of that path, equal to the sum of weights of its edges.

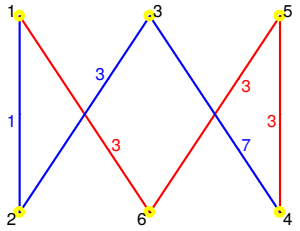
```
> G:=graph(%{[[1,2],1],[[1,6],3],[[2,3],3],[[3,4],7],[[4,5],3],[[5,6],3]%})
```

an undirected weighted graph with 6 vertices and 6 edges

```
> res:=dijkstra(G,1,4)
```

$$[[1,6,5,4],9]$$

```
> draw_graph(highlight_trail(G,res[0]))
```




```
> dijkstra(G,1)
```

$$[[1],0],[[1,2],1],[[1,6],3],[[1,2,3],4],[[1,6,5,4],9],[[1,6,5],6]$$

5.2.3. Traveling salesman problem

The command `traveling_salesman` is used for solving traveling salesman problem[5.1] (TSP) for undirected graphs.

`traveling_salesman` accepts the following arguments:

- an undirected graph $G(V,E)$,
- a weight matrix M (optional),
- a sequence of options (optional). Currently, the supported options are `approx` and `vertex_distance`.

5.1. For the details on traveling salesman problem and a historical overview see [6].

If the input graph G is unweighted and M is not specified, a Hamiltonian cycle (tour) is returned (the adjacency matrix of G is used for the edge weights). If G is weighted, two objects are returned: the optimal value for the traveling salesman problem and a Hamiltonian cycle which achieves the optimal value. If M is given and G is unweighted, M is used as the weight matrix for G.

If the option `vertex_distance` is passed and M is not specified, then for each edge $e \in E$ the Euclidean distance between its endpoints is used as the weight of e. Therefore it is required for each vertex in G to have a predefined position.

If the option `approx` is passed, a near-optimal tour is returned. In this case it is required that G is a complete weighted graph. For larger graphs, this is significantly faster than finding optimal tour. Results thus obtained are usually within just a few percent of the corresponding optimal values, despite the fact that the reported guarantee is generally much weaker (around 30%).

The strategy is to formulate TSP as a linear programming problem and to solve it by branch-and-cut method, applying the hierarchical clustering method of PFERSCHY and STANĚK [26] to generate subtour elimination constraints. The branching rule is implemented according to PADBERG and RINALDI [25]. In addition, the algorithm combines the method of CHRISTOFIDES [5], the method of farthest insertion and a simple variant of the powerful tour improvement heuristic developed by LIN and KERNIGHAN, implemented according to HELSGAUN [15], to generate near-optimal feasible solutions during the branch-and-cut process.

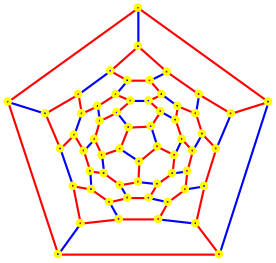
For Euclidean TSP instances, i.e. in cases when G is a complete graph with vertex distances as the edge weights, the algorithm usually finishes in less than a minute for problems with up to, say, 42 cities. For problems with 100 or more cities, the option `approx` is recommended as finding the optimal value may take a long time. Note that TSP is NP-hard, meaning that no polynomial time algorithm is known. Hence the algorithm may take exponential time to find the optimum in some instances, even the small ones (25-30 cities).

The following example demonstrates finding a Hamiltonian cycle in the truncated icosahedral ("soccer ball") graph. The result is visualized by using the `highlight_trail` command.

```
> G:=graph("soccerball")
```

an undirected unweighted graph with 60 vertices and 90 edges

```
> draw_graph(highlight_trail(G,traveling_salesman(G)),labels=false)
```



A matrix may be passed alongside an undirected graph to specify the edge weights. The alternative is to pass a weighted graph as the single argument.

```
> G:=graph("octahedron")
```

an undirected unweighted graph with 6 vertices and 12 edges

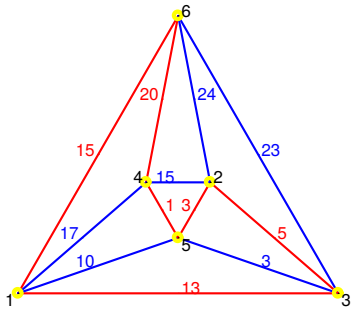
```
> M:=randmatrix(6,6,25)
```

$$\begin{pmatrix} 7 & 13 & 15 & 10 & 17 & 6 \\ 17 & 7 & 23 & 3 & 17 & 5 \\ 15 & 24 & 19 & 15 & 20 & 24 \\ 3 & 16 & 10 & 18 & 1 & 3 \\ 9 & 20 & 9 & 2 & 19 & 15 \\ 17 & 8 & 19 & 20 & 15 & 15 \end{pmatrix}$$

```
> c,t:=traveling_salesman(G,M)
```

$$57.0, [4, 5, 2, 3, 1, 6, 4]$$

```
> draw_graph(highlight_trail(make_weighted(G,M),t))
```



In the next example, an instance of Euclidean TSP with 42 cities is solved to optimality. The vertex positions are pairs of integers randomly chosen on the grid $[0, 1000] \times [0, 1000] \in \mathbb{Z}^2$.

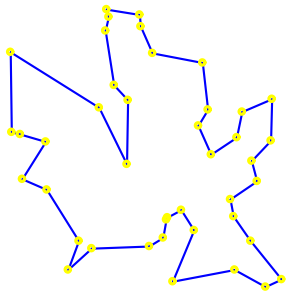
```
> G:=set_vertex_positions(complete_graph(42),[randvector(2,1000)$(k=1..42)])
```

an undirected unweighted graph with 42 vertices and 861 edges

```
> c,t:=traveling_salesman(G,vertex_distance):;
```

10.01 sec

```
> draw_graph(subgraph(G,trail2edges(t)),labels=false)
```



For large instances of Euclidean TSP the `approx` option may be used, as in the following example with 555 cities.

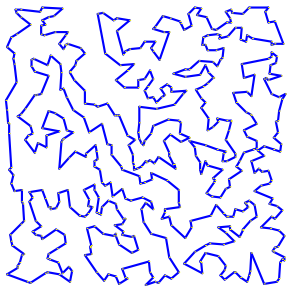
```
> H:=set_vertex_positions(complete_graph(555),[randvector(2,10000)$(k=1..555)])
```

an undirected unweighted graph with 555 vertices and 153735 edges

```
> ac,t:=traveling_salesman(H,vertex_distance,approx):;
```

49.34 sec

```
> draw_graph(subgraph(H,trail2edges(t)))
```



Near-optimal tours produced by the `approx` option are usually only slightly more expensive than the optimal ones. For example, a sub-optimal tour for the graph G with 42 vertices is obtained by the following command.

```
> ac,st:=traveling_salesman(G,vertex_distance,approx):;
```

```
The tour cost is within 28% of the optimal value
```

Although it is guaranteed that the near-optimal cost `ac` is for at most 28% larger than `c` (the optimum), the actual difference is within 3% of the latter:

```
> 100*(ac-c)/c
```

2.7105821877

5.3. SPANNING TREES

5.3.1. Constructing a spanning tree

The command `spanning_tree` is used for constructing a spanning tree of an undirected graph.

`spanning_tree` accepts one or two arguments, the input graph $G(V, E)$ and optionally a vertex $r \in V$. It returns the spanning tree T (rooted in r) of G, obtained by depth-first traversal in $O(|V| + |E|)$ time.

5.3.2. Minimal spanning tree

The command `minimal_spanning_tree` is used for constructing minimal spanning tree of an undirected graph.

`minimal_spanning_tree` accepts the input graph $G(V, E)$ as its only argument and returns its minimal spanning tree obtained by KRUSKAL's algorithm in $O(|E| \log |V|)$ time.

5.3.3. Counting all spanning trees

The command `number_of_spanning_trees` is used for counting all spanning trees in a graph.

`number_of_spanning_trees` accepts the input graph $G(V, E)$ as its only argument and returns the total number n of mutually different spanning trees in G.

The strategy is based on Theorem 2.2.12 (Matrix Tree Theorem) in [35], page 86. First the adjacency matrix A and the degree sequence δ of G are obtained. Then the matrix $B = \Delta - A$ is formed, where Δ is the diagonal matrix of order $|V|$ corresponding to δ. The last row and the last column of B are subsequently deleted, yielding the square matrix C of order $|V| - 1$. Now $n = \det C$.

```
> number_of_spanning_trees(graph("octahedron"))
```

384

```
> number_of_spanning_trees(graph("dodecahedron"))
```

5184000

```
> number_of_spanning_trees(hypercube_graph(4))
```

42467328

```
> number_of_spanning_trees(graph("soccerball"))
```

375291866372898816000

Chapter 6

Visualizing graphs

6.1. Drawing graphs by using various methods

To visualize a graph use the `draw_graph` command. It is capable to produce a drawing of the given graph using one of the several built-in methods.

6.1.1. Overview

`draw_graph` accepts one or two arguments, the mandatory first one being the graph $G(V, E)$. This command assigns 2D or 3D coordinates to each vertex $v \in V$ and produces a visual representation of G based on these coordinates. The second (optional) argument is a sequence of options. Each option is one of the following:

- `labels=true` or `false`: controls the visibility of vertex labels and edge weights (by default `true`, i.e. the labels and weights are displayed)
- `spring`: draw the graph G using a multilevel force-directed algorithm
- `tree[=r` or `[r1,r2,...]]`: draw the tree or forest G, optionally specifying root nodes for each tree
- `bipartite`: draw the bipartite graph G keeping the vertex partitions separated
- `circle[=L]` or `convexhull[=L]`: draw the graph G by setting the *hull vertices* from list $L \subset V$ (assuming $L = V$ by default) on the unit circle and all other vertices in origin, subsequently applying a force-directed vertex placement algorithm to generate the layout while keeping the hull vertices fixed
- `planar` or `plane`: draw the planar graph G using a force-directed algorithm
- `plot3d`: draw the connected graph G as if the `spring` option was enabled, but with vertex positions in 3D instead of 2D
- any unassigned identifier P: when given, the vertex coordinates will be stored to it in form of a list

The style options `spring`, `tree`, `circle`, `planar` and `plot3d` cannot be mixed, i.e. at most one can be specified. The option `labels` may be combined with any of the style options. Note that edge weights will not be displayed when using `plot3d` option when drawing a weighted graph.

When no style option is specified, the algorithm first checks if the graph G is a tree or if it is bipartite, in which cases it is drawn accordingly. Otherwise, the graph is drawn as if the option `circle` was specified.

Tree, circle and bipartite drawings can be obtained in linear time with a very small overhead, allowing graphs to be drawn quickly no matter the size. The force-directed algorithms are more expensive and operating in the time which is quadratic in the number of vertices. Their performance is, nevertheless, practically instantaneous for graphs with several hundreds of vertices (or less).

6.1.2. Drawing disconnected graphs

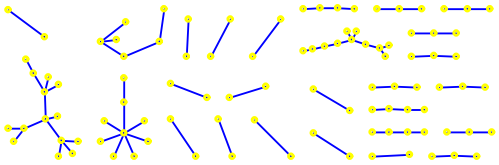
When the input graph has two or more connected components, each component is drawn separately and the drawings are subsequently arranged such that the bounding box of the whole drawing has the smallest perimeter under condition that as little space as possible is wasted inside the box.

For example, the command lines below draw a sparse random planar graph.

```
> G:=random_planar_graph(100,0.9,0)
```

an undirected unweighted graph with 100 vertices and 74 edges

```
> draw_graph(G,planar)
```



6.1.3. Spring method

When the option `spring` is specified, the input graph is drawn using the force-directed algorithm described in [19] (for an example of such a drawing see Figure 3.1). The idea, originally due to FRUCHTERMAN and REINGOLD [13], is to simulate physical forces in a spring-electrical model where the vertices and edges represent equally charged particles and springs connecting them, respectively.

In a spring-electrical model, each vertex is being repulsed by every other vertex with a force inversely proportional to the distance between them. At the same time, it is attracted to each of its neighbors with a force proportional to the square of the distance. Assuming that x_v is the vector representing the position of the vertex $v \in V$, the total force F_v applied to v is equal to

$$F_v = \sum_{w \in V \setminus \{v\}} -\frac{CK^2}{\|x_v - x_w\|^2}(x_v - x_w) + \sum_{w \in N(v)} \frac{\|x_v - x_w\|}{K}(x_v - x_w),$$

where $N(v)$ is the set of neighbors of v and C, K are certain positive real constants (actually, K may be any positive number, it affects only the scaling of the entire layout). Applying the forces iteratively and updating vertex positions in each iteration (starting from a random layout) leads the system to the state of minimal energy. By applying a certain "cooling" scheme to the model which cuts down the force magnitude in each iteration. the layout "freezes" after a number of iterations large enough to achieve the minimal energy state.

The force-directed method is computationally expensive and for larger graphs the pleasing layout cannot be obtained most of the time since the algorithm, starting with a random initial layout, gets easily "stuck" in the local energy minimum (ideally, the vertex positions should settle in the global minimal energy constellation). To avoid this a *multilevel* scheme is applied. The input graph is iteratively coarsened, either by removing the vertices from a maximal independent vertex set or contracting the edges of a maximal matching in each iteration. Each coarsening level is then processed by the force-directed algorithm, starting from the deepest (coarsest) one and *lifting* the obtained layout to the first upper level, using it as the initial layout for that level. The lifting is achieved by using the prolongation matrix technique described in [20]. To support drawing of large graphs (with 1000 vertices or more), the matrices used in the lifting process are stored as sparse matrices. The multilevel algorithm is significantly faster than the original, single-level one and usually produces better results.

Graph layouts obtained by using force-directed method have a unique property of reflecting symmetries in the design of the input graph, if any. Thus the drawings become more appealing and illustrate the certain properties of the input graph better. To make the symmetry more prominent, the drawing is rotated such that the axis, with respect to which the layout exhibits the largest *symmetry score*, becomes vertical. As the symmetry detection is in general very computationally expensive—up to $O(|V|^7)$ when using the symmetry measure of PURCHASE [34], for example—the algorithm deals only with the convex hull and the barycenter of the layout, which may not always be enough to produce the optimal result. Nevertheless, this approach is very fast and seems to work most of the time for graphs with a high level of symmetry (for example the octahedral graph).

For example, the following command lines produce a drawing of the tensor product of two graphs using the force-directed algorithm.

```
> G1:=graph(trail(1,2,3,4,5,2))
```

an undirected unweighted graph with 5 vertices and 5 edges

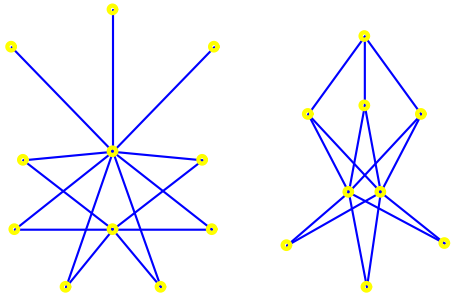
```
> G2:=star_graph(3)
```

an undirected unweighted graph with 4 vertices and 3 edges

```
> G:=tensor_product(G1,G2)
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> draw_graph(G,spring,labels=false)
```

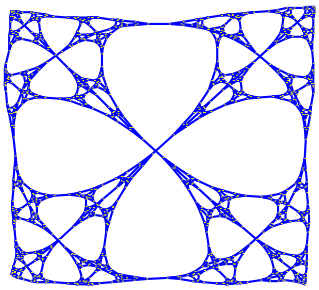


The following command lines demonstrate drawing of a much larger graph.

```
> S:=sierpinski_graph(5,4)
```

an undirected unweighted graph with 1024 vertices and 2046 edges

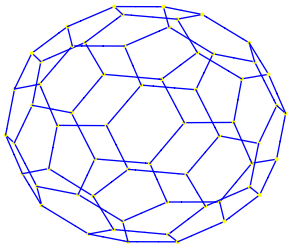
```
> draw_graph(S,spring)
```



Note that vertex labels are automatically suppressed because of the large number of vertices. On our system, the algorithm took less than 2 seconds to produce the layout.

The spring method is also used for creating 3D graph layouts, which are obtained by passing the option `plot3d` to the `draw_graph` command.

```
> draw_graph(graph("soccerball"),plot3d,labels=false)
```



```
> G1:=graph("icosahedron"):; G2:=graph("dodecahedron"):;
```

Done, Done

```
> G1:=highlight_edges(G1,edges(G1),red)
```

an undirected unweighted graph with 12 vertices and 30 edges

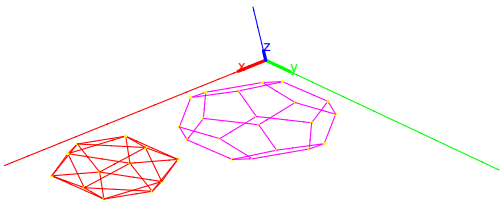
```
> G2:=highlight_edges(G2,edges(G2),magenta)
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> G:=disjoint_union(G1,G2)
```

an undirected unweighted graph with 32 vertices and 60 edges

```
> draw_graph(G,plot3d,labels=false)
```



6.1.4. Drawing trees

When the `tree[=r]` option is specified and the input graph G is a tree (and $r \in V$), it is drawn using a fast but simple node positioning algorithm inspired by the well-known algorithm of WALKER [33], using the first vertex (or the vertex r) as the root node. When drawing a rooted tree, one usually requires the following aesthetic properties [4].

A1. The layout displays the hierarchical structure of the tree, i.e. the y-coordinate of a node is given by its level.

A2. The edges do not cross each other.

A3. The drawing of a sub-tree does not depend on its position in the tree, i.e. isomorphic sub-trees are drawn identically up to translation.

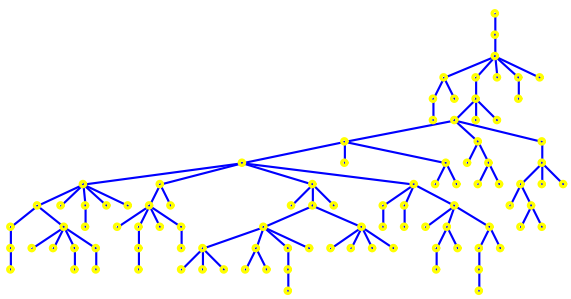
A4. The order of the children of a node is displayed in the drawing.

A5. The algorithm works symmetrically, i.e. the drawing of the reflection of a tree is the reflected drawing of the original tree.

The algorithm implemented in Giac generally satisfies all the above properties but A3. Instead, it tries to spread the inner sub-trees evenly across the available horizontal space. It works by organizing the structure of the input tree into levels by using depth-first search and laying out each level subsequently, starting from the deepest one and climbing up to the root node. In the end, another depth-first traversal is made, shifting the sub-trees horizontally to avoid intersections between their edges. The algorithm runs in $O(|V|)$ time and uses the minimum of horizontal space to draw the tree with respect to the specified root node r.

For example, the following command line produces the drawing of a random tree on 100 nodes.

```
> draw_graph(random_tree(100))
```



6.1.5. Drawing planar graphs

The algorithm implemented in Giac which draws planar graphs uses augmentation techniques to extend the input graph G to a graph G', which is homeomorphic to some triconnected graph, by adding temporary edges. The augmented graph G' is then drawn using Tutte's barycentric method [32], a force-directed algorithm which puts each vertex in the barycenter of its neighbors. It is guaranteed that a (non-strict) convex drawing will be produced, without edge crossings. In the end, the duplicate of the outer face and the temporary edges inserted during the augmentation stage are removed.

Tutte's algorithm requires that the vertices of the chosen outer face are initially fixed somewhere the boundary of a convex polygon. In addition, to produce a more flexible layout, the outer face is duplicated such that the subgraph induced by the vertices on both the outer face and its duplicate is a prism graph. Then only the duplicates of the outer face vertices are fixed, allowing the outer face itself to take a more natural shape. The duplicate of the outer face is removed after a layout is produced.

The augmentation process consists of two parts. Firstly, the input graph G is decomposed into biconnected components called *blocks* using the depth-first search (see [14], page 25). Each block is then decomposed into faces (represented by cycles of vertices) using Demoucron's algorithm (see [14], page 88, with a correction proposed in [23]). Embeddings obtained for each blocks are then combined by adding one temporary edge for each articulation point, joining the two corresponding blocks. Figure 6.1 shows the outer faces of two blocks B_1 and B_2, connected by an articulation point (cut vertex). The temporary edge (shown in blue) is added to join B_1 and B_2 into a single block. After "folding up" the tree of blocks, the algorithm picks the largest face in the resulting biconnected graph to be the outer face of the planar embedding.

The second part of the augmentation process consists of recursively decomposing each non-convex inner face into several convex polygons by adding temporary edges. An inner face $f=(v_1,...,v_n)$ is non-convex if there exist k and l such that $1\leqslant k<l-1<n$ and either $v_k\,v_l\in E$, in which case the edge $v_k\,v_l$ is a *chord* (see Figure 6.2 for an example) or there exists a face $g=(w_1,w_2,...,v_k,...,v_l,...,w_{m-1},w_m)$ such that the vertices $v_{k+1},...,v_{l-1}$ are not contained in g (see Figure 6.3 for an example). In Figures 6.1, 6.2 and 6.3, the temporary edges added by the algorithm are drawn in green.

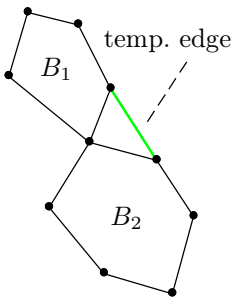


Fig. 6.1. joining two blocks

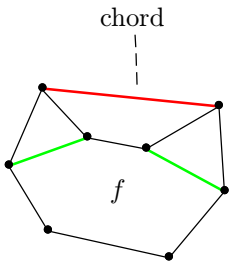


Fig. 6.2. a chorded face

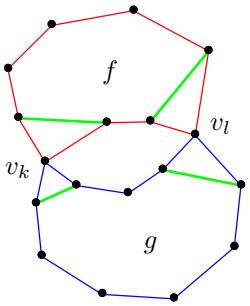


Fig. 6.3. two adjacent faces

This method of drawing planar graphs operates in $O(|V|^2)$ time. Nevertheless, it is quite fast for graphs up to 1000 vertices, usually producing results in less than a second. The drawback of this method is that it sometimes creates clusters of vertices which are very close to each other, resulting in a very high ratio of the area of the largest inner face to the area of the smallest inner face. However, if the result is not satisfactory, one should simply redraw the graph and repeat the process until a better layout is found. The planar embedding will in general be different each time.

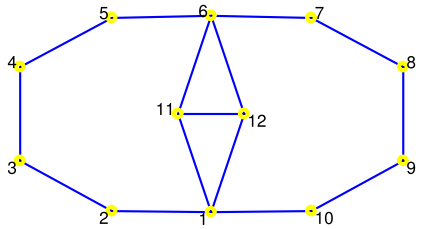
Another drawback of this method is that sparse planar graphs are sometimes drawn poorly.

The following example shows that the above described improvement of the barycentric method handles non-triconnected graphs well.

```
> G:=graph(trail(1,2,3,4,5,6,7,8,9,10,1),trail(11,12,6,11,1,12))
```

an undirected unweighted graph with 12 vertices and 15 edges

```
> draw_graph(G,planar)
```



Note that the inner diamond-like shape in the above drawing would end up flattened—making the two triangular faces invisible—if the input graph was not augmented. It is so because the vertices with labels 11 and 12 are "attracted" to each other (namely, the two large faces are "inflating" themselves to become convex), causing them to merge eventually.

In the following example the input graph G is connected but not biconnected (it has two articulation points). It is obtained by removing a vertex from the Sierpiński triangle graph ST_3^3. Note that the syntax mode is set to Xcas in this example, so the first vertex label is zero.

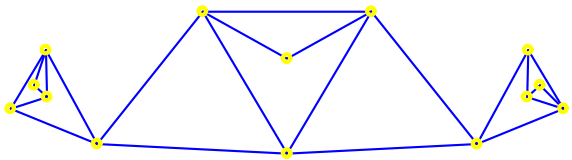
```
> G:=sierpinski_graph(3,3,triangle)
```

an undirected unweighted graph with 15 vertices and 27 edges

```
> G:=delete_vertex(G,3)
```

an undirected unweighted graph with 14 vertices and 23 edges

```
> draw_graph(G,planar,labels=false)
```



In the above example, several redraws were required to obtain a good planar embedding.

6.1.6. Circular graph drawings

The drawing method selected by specifying the option `circle=L` or `convexhull=L` when calling `draw_graph` on a triconnected graph $G(V, E)$, where $L \subset V$ is a set of vertices in G, uses the following strategy. First, positions of the vertices from L are fixed so that they form a regular polygon on the unit circle. Other vertices, i.e. all vertices from $V \setminus L$, are placed in origin. Then an iterative force-directed algorithm [27], similar to TUTTE's barycentric method, is applied to obtain the final layout.

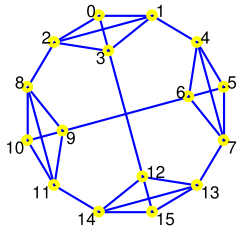
This approach gives best results for symmetrical graphs such as generalized Petersen graphs. In addition, if the input graph is planar, the drawing will also be planar (there is a possibility, however, that some very short edges may cross each other as the number of force update iterations is limited).

In the following example the Sierpiński graph S_4^2 is drawn using the above method. Note that the command lines below are executed in Xcas mode.

```
> G:=sierpinski_graph(2,4)
```

an undirected unweighted graph with 16 vertices and 30 edges

```
> draw_graph(G,circle=[0,1,4,5,7,13,15,14,11,10,8,2])
```



6.2. Custom vertex positions

6.2.1. Setting vertex positions

The command `set_vertex_positions` is used to assign custom coordinates to vertices of a graph to be used when drawing the graph.

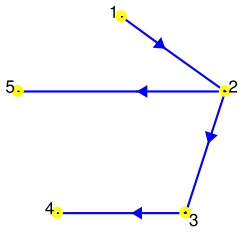
`set_vertex_positions` accepts two arguments, the graph $G(V,E)$ and the list L of positions to be assigned to vertices in order of `vertices(G)`. The positions may be complex numbers, lists of coordinates or points (geometrical objects created with the command `point`). `set_vertex_positions` returns the copy G' of G with the given layout stored in it.

Any subsequent call to `draw_graph` with G' as an argument and without specifying the drawing style will result in displaying vertices at the stored coordinates. However, if a drawing style is specified, the stored layout is ignored (although it stays stored in G').

```
> G:=digraph([1,2,3,4,5],%{[1,2],[2,3],[3,4],[2,5]%})
```

a directed unweighted graph with 5 vertices and 4 arcs

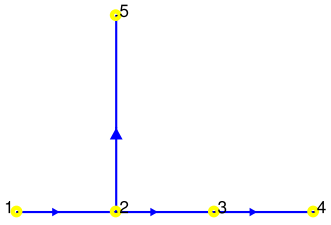
```
> draw_graph(G,circle)
```




```
> H:=set_vertex_positions(G,[[0,0],[0.5,0],[1.0,0],[1.5,0],[0.5,1]])
```

a directed unweighted graph with 5 vertices and 4 arcs

```
> draw_graph(H)
```



6.2.2. Generating vertex positions

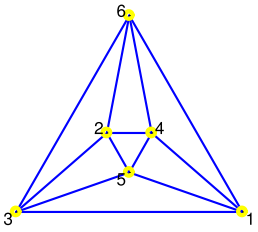
Vertex positions can be generated for a particular graph G by using the `draw_graph` command with the additional argument P which should be an unassigned identifier. After the layout is obtained, it will be stored to P as a list of positions (complex numbers for 2D drawings or points for 3D drawings) for each vertex in order of `vertices(G)`.

This feature combines well with the set_vertex_positions command, as when one obtains the desired drawing of the graph G by calling draw_graph, the layout coordinates can be easily stored to the graph for future reference. In particular, each subsequent call of draw_graph with G as an argument will display the stored layout. The example below illustrates this property by setting a custom layout to the octahedral graph.

```
> G:=graph("octahedron")
```

an undirected unweighted graph with 6 vertices and 12 edges

```
> draw_graph(G)
```



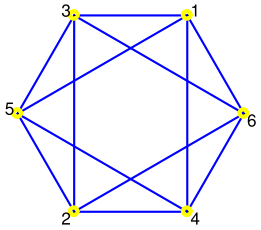

```
> draw_graph(G,P,spring):;
```

Now P contains vertex coordinates, which can be permanently stored to G:

```
> G:=set_vertex_positions(G,P)
```

an undirected unweighted graph with 6 vertices and 12 edges

```
> draw_graph(G)
```



It should be noted that, after a particular layout is fixed, it stays valid when some edges or vertices are removed or when an edge is contracted. The stored layout becomes invalid only if a new vertex is added to the graph (unless its position is specified by set_vertex_attribute upon the creation) or if the position attribute of an existing vertex is discarded.

6.3. HIGHLIGHTING PARTS OF A GRAPH

6.3.1. Highlighting vertices

The command highlight_vertex is used for changing color of one or more vertices in a graph.

highlight_vertex accepts two or three arguments: the graph $G(V, E)$, a vertex $v \in V$ or a list $L \subset V$ of vertices and optionally the new color (or a list of colors) for the selected vertices (the default color is green). It returns a modified copy of G in which the specified vertices are colored with the specified color.

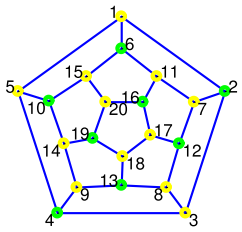
```
> G:=graph("dodecahedron")
```

an undirected unweighted graph with 20 vertices and 30 edges

```
> L:=maximum_independent_set(G)
```

$$[2, 4, 6, 12, 13, 10, 16, 19]$$

```
> draw_graph(highlight_vertex(G,L))
```



6.3.2. Highlighting edges and trails

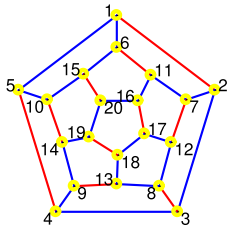
To highlight an edge or a set of edges in a graph, use the `highlight_edges` command. If the edges form a trail, it is usually more convenient to use the `highlight_trail` command (see below).

`highlight_edges` accepts two or three arguments: the graph $G(V, E)$, an edge e or a list of edges L and optionally the new color (or a list of colors) for the selected edges (the default color is red). It returns a modified copy of G in which the specified edges are colored with the specified color.

```
> M:=maximum_matching(G)
```

$$\{[1, 2], [5, 4], [6, 11], [3, 8], [7, 12], [9, 13], [10, 15], [14, 19], [16, 17]\}$$

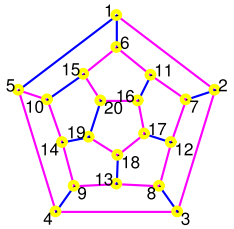
```
> draw_graph(highlight_edges(G,M))
```




```
> S:=spanning_tree(G)
```

an undirected unweighted graph with 20 vertices and 19 edges

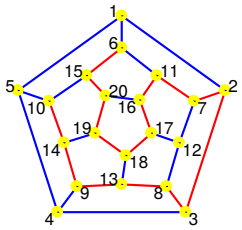
```
> draw_graph(highlight_edges(G,edges(S),magenta))
```



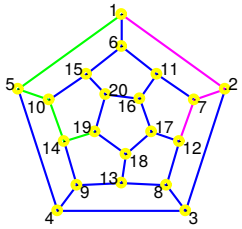
`highlight_trail` accepts two or three arguments: the input graph $G(V, E)$, a list $L \subset V$ or a list `[L1,L2,...,Lk]` of such lists (each list represents the vertices of the corresponding trail) and optionally a positive integer c or a list of positive integers `[c1,c2,...,ck]`. The command returns the copy of G in which edges between consecutive vertices in L are highlighted with color c (by default red) or the trail represented by L_i is highlighted with color c_i for $i = 1, 2, ..., k$.

Note that a trail can cross itself, which means that the elements of L are not required to be unique.

```
> draw_graph(highlight_trail(G,[6,15,20,19,18,17,16,11,7,2,3,8,13,9,14,10]))
```




```
> draw_graph(highlight_trail(G,shortest_path(G,1,[19,12]),[green,magenta]))
```



6.3.3. Highlighting subgraphs

The command `highlight_subgraph` is used for highlighting subgraph(s) of the given graph.

`highlight_subgraph` accepts two or four arguments: the graph $G(V, E)$, a subgraph S of G or a list of subgraphs of G and optionally the new colors for edges and vertices of the selected subgraph(s), respectively. It returns a modified copy of G with the selected subgraph(s) colored as specified.

```
> G:=graph(%{[1,2],[2,3],[3,1],[3,4],[4,5],[5,6],[6,4]%})
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> A:=articulation_points(G)
```

$[3, 4]$

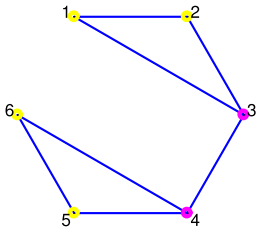
```
> B:=biconnected_components(G)
```

$[[4, 6, 5], [3, 4], [1, 3, 2]]$

```
> H:=highlight_vertex(G,A,magenta)
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> draw_graph(H)
```



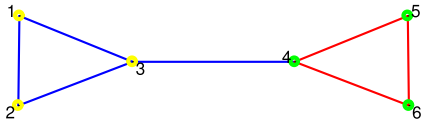

```
> S:=induced_subgraph(G,B[0])
```

an undirected unweighted graph with 3 vertices and 3 edges

```
> H:=highlight_subgraph(G,S)
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> draw_graph(H,spring)
```


1
2
3
4
5
6

Bibliography

[1] Mohsen Bayati, Jeong Han Kim, and Amin Saberi. A Sequential Algorithm for Generating Random Graphs. *Algorithmica*, 58:860–910, 2010.

[2] Danilo Blanuša. Problem četiriju boja. *Glasnik Mat.-Fiz. Astr. Ser. II*, 1:31–32, 1946.

[3] Daniel Brélaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22:251–256, 1979.

[4] Cristoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving Walker's Algorithm to Run in Linear Time. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing 2002, Lecture Notes in Computer Science vol 2528*, pages 344–353. Springer-Verlag Berlin Heidelberg, 2002.

[5] Nicos Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Report 388, Graduate School of Industrial Administration, 1976.

[6] William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[7] Melissa DeLeon. A Study of Sufficient Conditions for Hamiltonian Cycles. *Rose-Hulman Undergraduate Mathematics Journal*, 1, Article 6, 2000. https://scholar.rose-hulman.edu/rhumj/vol1/iss1/6.

[8] Isabel M. Díaz and Paula Zabala. A Branch-and-Cut Algorithm for Graph Coloring. *Discrete Applied Mathematics*, 154:826–847, 2006.

[9] Reinhard Diestel. *Graph Theory*. Springer-Verlag, New York, 1997.

[10] Jack Edmonds. Paths, Trees, and Flowers. In Gessel I. and GC. Rota, editors, *Classic Papers in Combinatorics*, pages 361–379. Birkhäuser Boston, 2009. Modern Birkhäuser Classics.

[11] S. Even and R. E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

[12] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.

[13] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience*, 21:1129–1164, 1991.

[14] Alan Gibbons. *Algorithmic graph theory*. Cambridge University Press, 1985.

[15] Keld Helsgaun. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Math. Prog. Comp.*, 1:119–163, 2009.

[16] Carl Hierholzer. Ueber die möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6:30–32, 1873.

[17] Andreas M. Hinz, Sandi Klavžar, and Sara S. Zemljič. A survey and classification of Sierpiński-type graphs. *Discrete Applied Mathematics*, 217:565–600, 2017.

[18] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[19] Yifan Hu. Efficient and High Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10:37–71, 2005.

[20] Yifan Hu and Jennifer Scott. A Multilevel Algorithm for Wavefront Reduction. *SIAM Journal on Scientific Computing*, 23:1352–1375, 2001.

[21] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5:558–562, 1962.

[22] B. D. McKay and A. Piperno. Practical Graph Isomorphism, II. *J. Symbolic Computation*, 60:94–112, 2013.

[23] Wendy Myrwold and Willian Kocay. Errors in graph embedding algorithms. *Journal of Computer and System Sciences*, 77:430–438, 2011.

[24] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002.

[25] Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *SIAM Review*, 33:60–100, 1991.

[26] Ulrich Pferschy and Rostislav Staněk. Generating subtour elimination constraints for the TSP from pure integer solutions. *Central European Journal of Operations Research*, 25:231–260, 2017.

[27] Bor Plestenjak. An Algorithm for Drawing Planar Graphs. *Software: Practice and Experience*, 29:973–984, 1999.

[28] Angelika Steger and Nicholas C. Wormald. Generating random regular graphs quickly. *Combinatorics Probability and Computing*, 8:377–396, 1999.

[29] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Comp.*, 1:146–160, 1972.

[30] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26:690–715, 1979.

[31] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363:28–42, 2006.

[32] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, s3-13:743–767, 1963.

[33] John Q. Walker II. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20:685–705, 1990.

[34] E. Welch and S. Kobourov. Measuring Symmetry in Drawings of Graphs. *Computer Graphics Forum*, 36:341–351, 2017.

[35] Douglas B. West. *Introduction to Graph Theory*. Pearson Education, 2002.

Command Index