Graph theory package for Giac/Xcas

Reference manual

DRAFT, June 2018

Table of contents

Chapter 1

Introduction

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

Chapter 2

Constructing graphs

2.1. General graphs

There are two commands in Giac that allow construction of general graphs: `graph` and `digraph`.

2.1.1. 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. Clebsch graph: `clebsch`
2. Coxeter graph: `coxeter`
3. Desargues graph: `desargues`
4. Dodecahedral graph: `dodecahedron`
5. Dürer graph: `durer`
6. Dyck graph: `dyck`
7. Grinberg graph: `grinberg`
8. Grotzsch graph: `grotzsch`
9. Harries graph: `harries`
10. Harries–Wong graph: `harries-wong`
11. Heawood graph: `heawood`

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

2.1.2. 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.

2.1.3. 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

2.1.4. 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

2.1.5. 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

2.1.6. 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

2.2. PROMOTING TO DIRECTED AND UNDIRECTED GRAPHS

2.2.1. Converting edges to pairs of arcs

To promote an existing undirected graph to a directed one or an unweighted graph to a weighted one, use the commands `make_directed` and `make_weighted`, respectively.

The command `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

2.2.2. Assigning weight matrix to an unweighted graph

The command `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

2.3. CYCLE AND PATH GRAPHS

2.3.1. Cycle graphs

Cycle graphs can be created by using the command `cycle_graph`.

`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 $n=4$.

```
> cycle_graph(5)
```

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

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

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

2.3.2. Path graphs

Path graphs can be created by using the command `path_graph`.

`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 and 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

2.3.3. 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).

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

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

2.4. Complete graphs

2.4.1. Complete graphs with multiple vertex partitions

To create complete (multipartite) graphs, use the command `complete_graph`.

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 $n_1, n_2, ..., n_k$.

```
> complete_graph(5)
```

an undirected unweighted graph with 5 vertices and 10 edges

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

an undirected unweighted graph with 3 vertices and 3 edges

```
> complete_graph(2,3)
```

an undirected unweighted graph with 5 vertices and 6 edges

2.4.2. 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

2.5. SEQUENCE GRAPHS

2.5.1. Creating graphs from degree sequences

To construct a graph from its degree sequence, use the command `sequence_graph`.

`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.

2.5.2. Validating graphic sequences

The command `is_graphic_sequence` is used to check whether a list of integers represents the degree sequence of some graph.

`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, else it returns `false`. 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

2.6. Intersection graphs

2.6.1. Interval graphs

The command `interval_graph` is used for creating 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.

```
> 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

2.6.2. Kneser graphs

To create Kneser graphs use the commands `kneser_graph` and `odd_graph`.

`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 a certain intersection graph.

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

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

2.7. Special graphs

2.7.1. 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

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

an undirected unweighted graph with 32 vertices and 80 edges

2.7.2. 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.

```
> star_graph(5)
```

an undirected unweighted graph with 6 vertices and 5 edges

2.7.3. 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.

```
> wheel_graph(5)
```

an undirected unweighted graph with 6 vertices and 10 edges

2.7.4. 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)`.

```
> web_graph(7,3)
```

an undirected unweighted graph with 21 vertices and 35 edges

2.7.5. 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 `web_graph(n,2)`.

```
> prism_graph(5)
```

an undirected unweighted graph with 10 vertices and 15 edges

2.7.6. 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.

```
> antiprism_graph(5)
```

an undirected unweighted graph with 10 vertices and 20 edges

2.7.7. Grid graphs

The command `grid_graph` is used for creating rectangular 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)`.

```
> grid_graph(5,3)
```

an undirected unweighted graph with 15 vertices and 22 edges

To create torus grid graphs, use the command `torus_grid_graph` which accepts two positive integers m and n as its arguments and returns the m by n torus grid on $m\,n$ vertices, namely the Cartesian product of `cycle_graph(m)` and `cycle_graph(n)`.

```
> torus_grid_graph(5,3)
```

an undirected unweighted graph with 15 vertices and 30 edges

2.7.8. Sierpiński graphs

The command `sierpinski_graph` is used for creating Sierpiński-type graphs S_k^n and ST_k^n [6].

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

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

an undirected unweighted graph with 81 vertices and 120 edges

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

an undirected unweighted graph with 42 vertices and 81 edges

2.7.9. 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.

For example, 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:

```
> petersen_graph(8,3)
```

an undirected unweighted graph with 16 vertices and 24 edges

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

2.7.10. LCF graphs

2.8. Isomorphic copies of graphs

The commands `isomorphic_copy`, `permute_vertices` and `relabel_vertices` are used to obtain isomorphic copies of an existing graph.

2.8.1. Creating an isomorphic copy of a graph

The 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.

```
> isomorphic_copy(path_graph(5),randperm(5))
```

an undirected unweighted graph with 5 vertices and 4 edges

2.8.2. Permuting graph vertices

The command `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.

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

an undirected unweighted graph with 4 vertices and 3 edges

2.8.3. Relabeling graph vertices

The command `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.

```
> relabel_vertices(path_graph(4),[a,b,c,d])
```

an undirected unweighted graph with 4 vertices and 3 edges

2.9. Subgraphs

2.9.1. 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.

```
> subgraph(complete_graph(5),[[1,2],[2,3],[3,4],[4,1]])
```

an undirected unweighted graph with 4 vertices and 4 edges

2.9.2. 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.

```
> induced_subgraph(petersen_graph(5),[1,2,3,6,7,9])
```

an undirected unweighted graph with 6 vertices and 6 edges

2.9.3. 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 arcs and weights of edges/arcs.

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

a directed weighted graph with 3 vertices and 3 arcs

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

an undirected unweighted graph with 3 vertices and 3 edges

```
> edges(U)
```

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

2.10. Operations on graphs

2.10.1. 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.

```
> graph_complement(cycle_graph(5))
```

an undirected unweighted graph with 5 vertices and 5 edges

2.10.2. 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.

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

an undirected unweighted graph with 5 vertices and 7 edges

2.10.3. 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)
```

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

2.10.4. 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 $G(V, E)$ with $V = V_1 \cup V_2 \cup \cdots \cup V_k$ 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)
```

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

2.10.5. 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 $|E| = \sum_{k=1}^{n} |E_k|$.

```
> disjoint_union(cycle_graph(3),path_graph(3))
```

an undirected unweighted graph with 6 vertices and 5 edges

2.10.6. 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 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.

```
> graph_join(path_graph(2),graph(3))
```

an undirected unweighted graph with 5 vertices and 7 edges

2.10.7. 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.

```
> graph_power(path_graph(5),2)
```

an undirected unweighted graph with 5 vertices and 7 edges

```
> graph_power(path_graph(5),3)
```

an undirected unweighted graph with 5 vertices and 9 edges

2.10.8. 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`) is member of 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

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

an undirected unweighted graph with 4 vertices and 3 edges

```
> cartesian_product(G1,G2)
```

an undirected unweighted graph with 20 vertices and 35 edges

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`) is in E if and only if u_1 is adjacent to u_2 **and** v_1 is adjacent to v_2.

```
> tensor_product(G1,G2)
```

an undirected unweighted graph with 20 vertices and 30 edges

2.10.9. Transitive closure

2.10.10. Plane dual graph

2.11. RANDOM GRAPHS

2.11.1. Random general graphs

2.11.2. Random bipartite graphs

2.11.3. Random trees

2.11.4. Random planar graphs

2.11.5. Random regular graphs

2.11.6. Random tournaments

2.11.7. Random flow networks

2.11.8. Randomizing edge weights

Chapter 3

Modifying graphs

3.1. Vertices

3.1.1. 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

3.2. Edges

3.2.1. 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` and `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

3.2.2. 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

3.2.3. 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 $G'(V \setminus \{w\}, E')$.

```
> 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. For example, if G is the complete graph K_5 and $k = 2$, $e_1 = \{1, 2\}$ and $e_2 = \{3, 4\}$, input:

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

an undirected unweighted graph with 5 vertices and 10 edges

```
> foldl(contract_edge,K5,[1,2],[3,4])
```

an undirected unweighted graph with 3 vertices and 3 edges

3.2.4. 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.

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

an undirected unweighted graph with 3 vertices and 3 edges

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

an undirected unweighted graph with 4 vertices and 4 edges

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

an undirected unweighted graph with 5 vertices and 5 edges

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

an undirected unweighted graph with 6 vertices and 6 edges

3.3. Attributes

3.3.1. Graph attributes

The graph structure maintains a set of attributes as key-value pairs. The command `set_graph_attribute` is used to modify the existing values or add new attributes.

`set_graph_attribute` 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. Two tags are predefined and used by the CAS commands: `"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.

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.

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

a directed unweighted graph

```
> G:=set_graph_attribute(G,"name"="C3","shape"=triangle)
```

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

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

$'\text{triangle}'$

```
> list_graph_attributes(G)
```

$[\text{directed} = \text{true}, \text{weighted} = \text{false}, \text{name} = C3, \text{shape} = '\text{triangle}']$

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

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

```
> list_graph_attributes(G)
```

$[\text{directed} = \text{true}, \text{weighted} = \text{false}, \text{name} = C3]$

3.3.2. Vertex attributes

For every vertex of a graph, the list of attributes in form of key-value pairs is maintained. The command `set_vertex_attribute` is used to modify the existing values or to add new attributes.

`set_vertex_attribute` 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.

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

```
>
```

3.3.3. Edge attributes

For every edge of a graph, the list of attributes in form of key-value pairs is maintained. The command `set_edge_attribute` is used to change existing values or add new attributes.

`set_edge_attribute` 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.

```
>
```

Chapter 4

Import and export

4.1. Importing graphs

4.1.1. Loading graphs from dot files

The command `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 the `dot` format. The file extension `.dot` may be omitted in the `filename` since `dot` is the only supported format. 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 the details about the `dot` format see Section 4.1.2.

For example, assume that the file `philosophers.dot` is saved in the directory `Documents/dot/`, containing the graph describing the famous "dining philosophers" problem. To import it, input:

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

an undirected unweighted graph with 21 vertices and 27 edges

4.1.2. The dot file format overview

Giac has some basic support for the `dot` language[4.1]. 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? {
    ...
}
```

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.

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

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};
}
```

4.2. EXPORTING GRAPHS

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

4.2.1. 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_philosophers")
```

1

4.2.2. 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[4.2] 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
```

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

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 4.1.

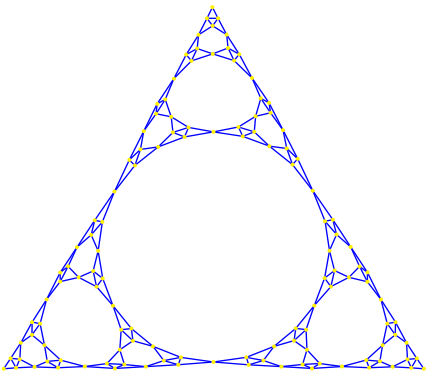


Fig. 4.1. drawing of the Sierpiński graph ST_3^5 using LaTeX and PSTricks

Chapter 5
Graph properties

5.1. Basic properties

5.1.1. Listing vertices and edges of a graph

The command `vertices` or `graph_vertices` resp. `edges` is used for extracting set of vertices resp. set of edges from a graph.

`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).

```
> 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)
```

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

```
> 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)
```

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

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

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

5.1.2. 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(1,2,3,4,2,5,6,3,7,8,3,9,1))
```

an undirected unweighted graph with 9 vertices and 12 edges

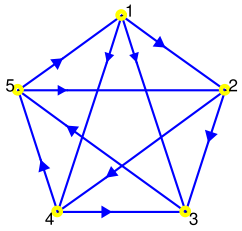
```
> vertex_degree(G,3)
```

6

```
> 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)
```

$[2, 4, 6, 2, 2, 2, 2, 2, 2]$

5.1.3. Adjacent vertices

```
>
```

5.1.4. Incident edges

5.2. ALGEBRAIC PROPERTIES

5.3. CONNECTIVITY

5.3.1. Vertex connectivity

5.3.2. Edge connectivity

5.3.3. Connected components

5.3.4. Strongly connected components

5.3.5. Articulation points

5.3.6. Bridges

5.3.7. Maximum matching in bipartite graphs

The algorithm [7]

```
>
```

5.4. Trees

5.4.1. Checking if graph is a tree

5.4.2. Checking if graph is a forest

5.4.3. Measuring the height of a tree

5.5. Cycles and paths

5.5.1. Computing the girth of a graph

5.5.2. Acyclic directed graphs

5.5.3. Eulerian graphs

5.5.4. Hamiltonian graphs

5.6. Matchings

5.6.1. Maximum matching

The command `maximum_matching` is used for finding maximum matching in undirected 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 incident for all $1 \leqslant i < j \leqslant m$, under condition that m is maximized. Edges e_k for $k = 1, ..., m$ represent matching pairs of vertices in G.

The command applies Edmonds' blossom algorithm [3], which finds maximum matching in $O(|E|\,|V|^2)$ time.

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

an undirected unweighted graph with 6 vertices and 12 edges

```
> maximum_matching(G)
```

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

5.7. Cliques

5.7.1. Checking if graph is a clique

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

5.7.2. Counting 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.

The strategy used to find all maximal cliques is a variant of the algorithm of BRON and KERBOSCH as described in [13]. Its worst-case running time is $O(3^{|V|/3})$. However, the performance is usually almost instantaneous 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)
```

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

237 msec

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

an undirected unweighted graph with 100 vertices and 2461 edges

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

3124

214 msec

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

an undirected unweighted graph with 500 vertices and 31250 edges

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

$$\begin{pmatrix} 5 & 148657 \\ 6 & 17834 \\ 7 & 356 \end{pmatrix}$$

1.254 sec

5.7.3. 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 in [11].

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

an undirected unweighted graph with 3125 vertices and 7810 edges

```
> maximum_clique(G)
```

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

231 msec

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

an undirected unweighted graph with 300 vertices and 13505 edges

```
> maximum_clique(G)
```

$[231, 225, 187, 239, 164, 292, 296]$

232 msec

5.7.4. Minimum clique cover

The *minimum clique cover* 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 $v\in C_i$. In Giac, such cover can be obtained by calling the `clique_cover` command.

`clique_cover` accepts graph G as its mandatory argument and returns the smallest possible cover. 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 cover 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 cover 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 cover 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(30,0.5)
```

an undirected unweighted graph with 30 vertices and 218 edges

```
> clique_cover(G)
```

```
Constructing conflict graph...
Conflict graph has 109 + 5 = 114 vertices
```

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

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

```
> clique_cover(graph("soccerball"))
```

[[1, 2], [5, 4], [6, 7], [3, 16], [11, 12], [21, 22], [26, 27], [10, 9], [8, 49], [30, 29], [53, 52], [15, 14], [13, 54], [58, 57], [17, 18], [20, 19], [59, 60], [38, 37], [25, 24], [23, 39], [43, 42], [28, 44], [48, 47], [31, 32], [35, 34], [36, 40], [33, 46], [41, 45], [51, 55], [56], [50]]

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)
```

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

5.8. 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 different color. The only requirement is that the colors of adjacent vertices must differ from one another.

5.8.1. Greedy coloring

To color vertices of a graph in a greedy fashion use the command `greedy_color`.

`greedy_color` accepts one mandatory argument, the 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) such that each vertex gets the smallest available color. The list of vertex colors is returned, following the order of `vertices(G)`.

Generally, different choices of permutation p produce different colorings. Also, 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]

```
> greedy_color(G,randperm(10))
```

[3, 2, 3, 2, 1, 2, 1, 1, 3, 2]

5.8.2. Minimal coloring

The vertex coloring of G is *minimal* 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 to solve the minimal vertex coloring problem (MVCP), which is converted to the equivalent integer linear programming problem as described in [2].

```
>
```

5.9. Edge coloring

5.10. Vertex ordering

Chapter 6
Traversing graphs

6.1. Optimal routing

6.1.1. 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,1,9)
```

$[1, 5, 4, 9]$

```
> shortest_path(G,1,[7,9])
```

$[[1, 2, 7], [1, 5, 4, 9]]$

6.1.2. Cheapest paths in weighted graphs

The command `dijkstra` is used to find the cheapest path between two 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.

```
>
```

6.1.3. Traveling salesman problem

6.2. Spanning trees

6.2.1. Spanning tree construction

The command `spanning_tree` accepts one or two arguments, an undirected graph G and optionally a vertex $r \in V$. It returns the spanning tree T of G rooted in r or, if none is given, in the first vertex in the list V, obtained by depth-first traversal in $O(|V| + |E|)$ time.

6.2.2. Minimal spanning tree

The command `minimal_spanning_tree` accepts an undirected 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.

Chapter 7

Visualizing graphs

7.1. Drawing graphs 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.

7.1.1. 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]`: 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 bipartite or a tree, in which case it is drawn accordingly. Else 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).

7.1.2. Drawing disconnected graphs

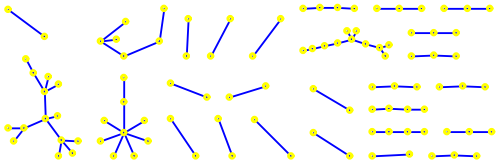
When the input graph has two or more connected components, each component is drawn separately and the drawings are subsequently arranged such that the bounding box of the whole drawing has the smallest perimeter under condition that as little space as possible is wasted inside the box.

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)
```



7.1.3. Spring method

When the option `spring` is specified, the input graph is drawn using the force-directed algorithm described in [8] (for an example of such a drawing see Figure 4.1). The idea, originally due to FRUCHTERMAN and REINGOLD [4], 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 force inversely proportional to the distance. At the same time, it is attracted to each of its neighbors with 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 [9] (to support drawing of large graphs with more than 1000 vertices, the matrices used in the lifting process are stored as sparse matrices). The multilevel algorithm is significantly faster than the original 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 [16], 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

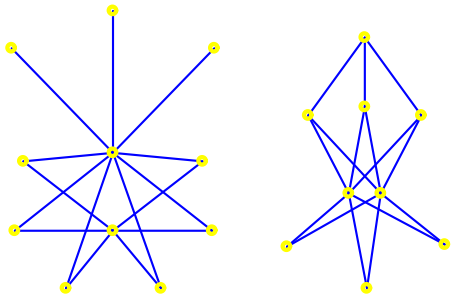
```
> G2:=star_graph(3)
```

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)
```

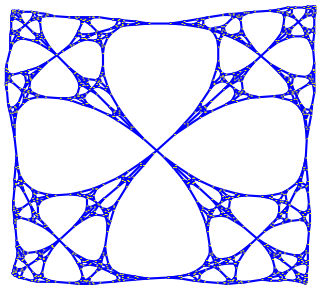


The following command lines demostrate drawing of a much larger graph.

```
> 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.

7.1.4. 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 [15], using the first vertex (or the vertex r) as the root node. When drawing a rooted tree, one usually requires the following aesthetic properties [1]:

A1. The layout displays the hierarhical 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 subtree does not depend on its position in the tree, i.e. isomorphic subtrees 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.

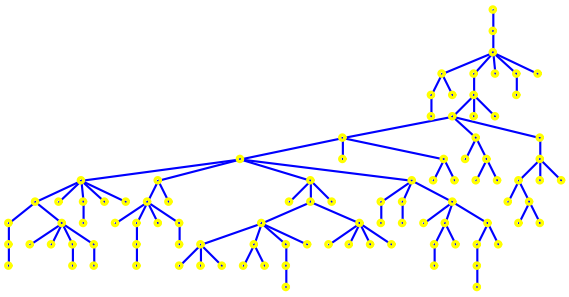
The algorithm implemented in Giac generally satisfies all the above properties but A3. Instead, it tries to spread the inner subtrees evenly across the available horizontal space. It works by organizing the structure of the input tree into levels by using depth-first search and laying out each level subsequently, starting from the deepest one and climbing up to the root node. In the end, another depth-first traversal is made, shifting the subtrees horizontally to avoid intersections between them. 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 lines produce the drawing of a random tree on 100 nodes.

```
> T:=random_tree(100)
```

an undirected unweighted graph with 100 vertices and 99 edges

```
> draw_graph(T)
```



7.1.5. 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 [14], 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 [5], page 25). Each block is then decomposed into faces (represented by cycles of vertices) using DEMOUCRON's algorithm (see [5], page 88, with a correction proposed in [10]). Embeddings obtained for each blocks are then combined by adding one temporary edge for each articulation point, joining the two corresponding blocks. Figure 7.1 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.

The second part of the augmentation process consists of decomposing each non-convex inner face into several convex polygons by adding temporary edges. An inner face $f=(v_1,\ldots,v_n)$ 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 7.2 for an example) or there exists a face $g=(w_1,w_2,\ldots,v_k,\ldots,v_l,\ldots,w_{m-1},w_m)$ such that the vertices $v_{k+1},\ldots,v_{l-1}$ are not contained in g (see Figure 7.3 for an example).

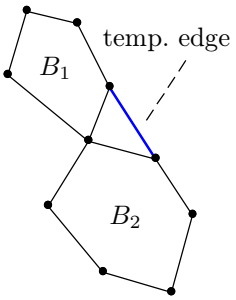


Fig. 7.1. joining two blocks

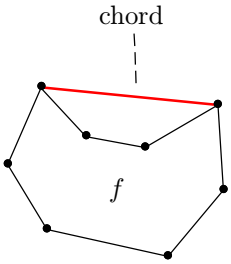


Fig. 7.2. a chorded face

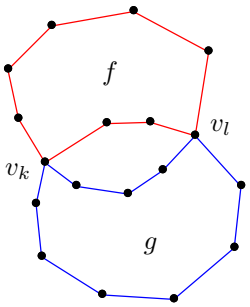


Fig. 7.3. two adjacent faces

This method of drawing planar graphs operates in $O(|V|^2)$ time. Nevertheless, it is very fast for graphs up to 1000 vertices, 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.

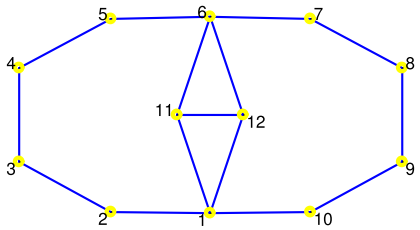
Another drawback of this method is that sparse planar graphs are sometimes drawn poorly.

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.

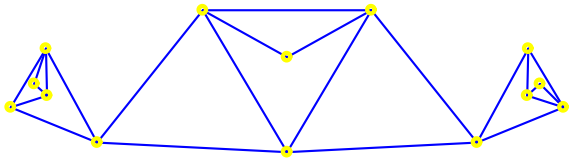
```
> G:=sierpinski_graph(3,3,triangle)
```

an undirected unweighted graph with 15 vertices and 27 edges

```
> G:=delete_vertex(G,8)
```

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.

7.1.6. Circular graph drawings

The drawing method selected by specifying the option `circle=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 [12], similar to TUTTE's barycentric method, is applied to obtain the final layout.

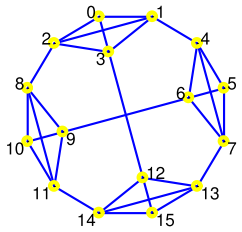
This approach produces pleasant drawings of round, symmetrical graphs, such as generalized Petersen graphs. In addition, if the input graph is planar, the drawing will also be planar (there is a possibility, however, that some very short edges may cross each other).

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])
```



7.2. CUSTOM VERTEX POSITIONS

7.2.1. 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.

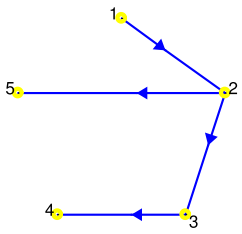
`set_vertex_positions` accepts two arguments, the graph $G(V,E)$ and the list L of positions to be assigned to vertices in order of `vertices(G)`. The positions may be complex numbers, lists of coordinates or points (geometrical objects created with the command `point`). `set_vertex_positions` returns 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

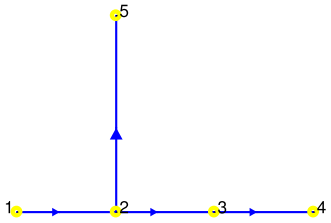
```
> draw_graph(G,circle)
```



```
> 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)
```



7.2.2. Generating vertex positions

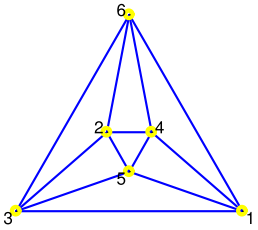
Vertex positions can be generated for a particular graph G by using the `draw_graph` command with the additional argument P which should be an unassigned identifier. After the layout is obtained, it will be stored to P as a list of positions (complex numbers for 2D drawings or points for 3D drawings) 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)
```



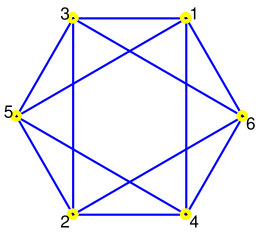

```
> draw_graph(G,P,spring):;
```

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. In the latter case the two endpoints collapse into a single vertex located at the midpoint of the contracted edge. The stored layout becomes invalid if new vertices are added to the graph, unless their positions are specified by `set_vertex_attribute` upon creation.

7.3. HIGHLIGHTING PARTS OF A GRAPH

7.3.1. 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 for the selected vertices (the default is green). It returns a modified copy of G in which the specified vertices are colored with the specified color.

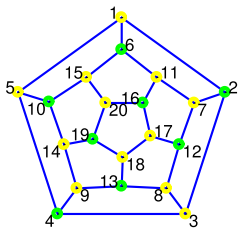
```
> G:=graph("dodecahedron")
```

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))
```



7.3.2. 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 call the `highlight_trail` command.

`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 for the selected edges (the default is red). It returns a modified copy of G in which the specified edges are colored with the specified color.

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

$$\begin{pmatrix} 1 & 2 \\ 5 & 4 \\ 6 & 11 \\ 3 & 8 \\ 7 & 12 \\ 9 & 13 \\ 10 & 14 \\ 15 & 20 \\ 16 & 17 \\ 18 & 19 \end{pmatrix}$$

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

The output is shown in Figure 7.4.

```
> S:=spanning_tree(G)
```

an undirected unweighted graph with 20 vertices and 19 edges

```
> draw_graph(highlight_edges(G,edges(S),magenta))
```

The output is shown in Figure 7.5.

```
> T:=[6,15,20,19,18,17,16,11,7,2,3,8,13,9,14,10,5,1,6]
```

$$[6, 15, 20, 19, 18, 17, 16, 11, 7, 2, 3, 8, 13, 9, 14, 10, 5, 1, 6]$$

```
> draw_graph(highlight_trail(G,T))
```

The output is shown in Figure 7.6.

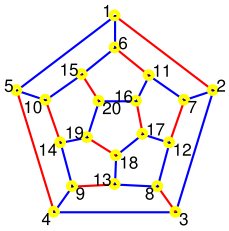


Fig. 7.4. maximum matching

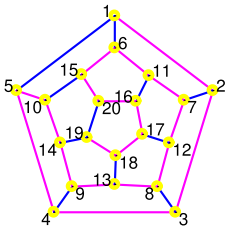


Fig. 7.5. spanning tree

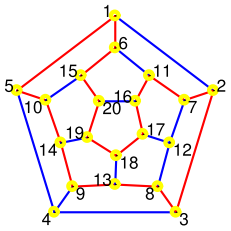


Fig. 7.6. cyclic path

7.3.3. 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)
```

$$[3, 4]$$

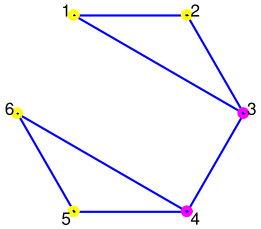
```
> B:=biconnected_components(G)
```

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

```
> H:=highlight_vertex(G,A,magenta)
```

an undirected unweighted graph with 6 vertices and 7 edges

```
> draw_graph(H)
```



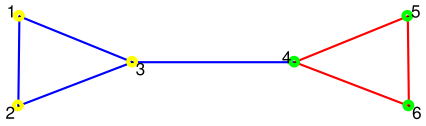

```
> S:=induced_subgraph(G,B[0])
```

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)
```



Bibliography

[1] 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.

[2] Isabel M. Díaz and Paula Zabala. A Branch-and-Cut Algorithm for Graph Coloring. *Discrete Applied Mathematics*, 154:826–847, 2006.

[3] 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.

[4] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience*, 21:1129–1164, 1991.

[5] Alan Gibbons. *Algorithmic graph theory*. Cambridge University Press, 1985.

[6] Andreas M. Hinz, Sandi Klavžar, and Sara S. Zemljič. A survey and classification of Sierpiński-type graphs. *Discrete Applied Mathematics*, 217:565–600, 2017.

[7] 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.

[8] Yifan Hu. Efficient and High Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10:37–71, 2005.

[9] Yifan Hu and Jennifer Scott. A Multilevel Algorithm for Wavefront Reduction. *SIAM Journal on Scientific Computing*, 23:1352–1375, 2001.

[10] Wendy Myrwold and Willian Kocay. Errors in graph embedding algorithms. *Journal of Computer and System Sciences*, 77:430–438, 2011.

[11] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002.

[12] Bor Plestenjak. An Algorithm for Drawing Planar Graphs. *Software: Practice and Experience*, 29:973–984, 1999.

[13] 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.

[14] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, s3-13:743–767, 1963.

[15] John Q. Walker II. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20:685–705, 1990.

[16] E. Welch and S. Kobourov. Measuring Symmetry in Drawings of Graphs. *Computer Graphics Forum*, 36:341–351, 2017.