Graph theory package for Giac/Xcas

User manual

Luka Marohnić

June 8, 2018

Contents

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.

The commands are divided into the following six sections: *Constructing graphs*, *Modifying graphs*, *Import and export*, *Graph properties*, *Traversing graphs* and *Visualizing graphs*.

2 Constructing graphs

2.1 Creating graphs from scratch: `graph`, `digraph`

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.

- Clebsch graph: `clebsch`
- Coxeter graph: `coxeter`
- Desargues graph: `desargues`
- Dodecahedral graph: `dodecahedron`
- Dürer graph: `durer`
- Dyck graph: `dyck`
- Grinberg graph: `grinberg`
- Grotzsch graph: `grotzsch`
- Harries graph: `harries`
- Harries–Wong graph: `harries-wong`
- Heawood graph: `heawood`
- Herschel graph: `herschel`
- Icosahedral graph: `icosahedron`
- Levi graph: `levi`

- Ljubljana graph: `ljubljana`
- McGee graph: `mcgee`
- Möbius–Kantor graph: `mobius-kantor`
- Nauru graph: `nauru`
- Octahedral graph: `octahedron`
- Pappus graph: `pappus`
- Petersen graph: `petersen`
- Robertson graph: `robertson`
- Soccer ball graph: `soccerball`
- Shrikhande graph: `shrikhande`
- Tetrahedral graph: `tehtrahedron`

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 demostrates the various possibilities when using `graph` and `digraph` commands.

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.
Input:

```
graph(5)
```

Output:

```
an undirected unweighted graph with 5 vertices and 0 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 3 vertices and 0 edges
```

Creating single 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.
Input:

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

Output:

```
an undirected unweighted graph with 3 vertices and 3 edges
```

Edge weights may also be specified.
Input:

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

Output:

```
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.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 3 edges
```

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.
Input:

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

Output:

```
a directed unweighted graph with 4 vertices and 3 arcs
```

Alternatively, one may specify edges as a trail.
Input:

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

Output:

```
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.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 3 edges
```

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

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

Output:

```
an undirected unweighted graph with 7 vertices and 9 edges
```

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.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 3 edges
```

Input:

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

Output:

```
a directed weighted graph with 4 vertices and 4 arcs
```

List of vertex labels can be specified before the matrix.
Input:

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

Output:

```
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.
Input:

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

Output:

```
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.
Input:

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

Output:

```
an undirected unweighted graph with 10 vertices and 15 edges
```

2.2 Promoting to directed and/or weighted graphs: `make_directed`, `make_weighted`

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.
Input:

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

Output:

```
a directed unweighted graph with 4 vertices and 8 arcs
```

Input:

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

Output:

```
a directed weighted graph with 4 vertices and 8 arcs
```

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.
Input:

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

Output:

```
an undirected weighted graph with 3 vertices and 3 edges
```

2.3 Cycle graphs: `cycle_graph`

The command `cycle_graph` accepts a positive integer n or a list of distinct vertices as its only argument and returns a 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).
Input:

```
cycle_graph(5)
```

Output:

```
an undirected unweighted graph with 5 vertices and 5 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 5 edges
```

2.4 Path graphs: `path_graph`

The command `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).
Input:

```
path_graph(5)
```

Output:

```
an undirected unweighted graph with 5 vertices and 4 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 4 edges
```

2.5 Trail of edges: `trail`

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

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

Output:

```
trail(1,2,3,4,2), an undirected unweighted graph with 4
                  vertices and 4 edges
```

2.6 Complete graphs: `complete_graph`, `complete_binary_tree`, `complete_kary_tree`

The command `complete_graph` creates complete (multipartite) graphs. It can be called with a single argument, a positive integer n or a list of distinct vertices, in which case the complete graph with the specified vertices will be returned. 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). Alternatively, a sequence of positive integers $n_1, n_2, \ldots, n_k$ may be passed as arguments, in which case the complete multipartite graph with partitions of size $n_1, n_2, \ldots, n_k$ will be returned.
Input:

```
complete_graph(5)
```

Output:

```
an undirected unweighted graph with 5 vertices and 10 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 3 vertices and 3 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 6 edges
```

The command `complete_binary_tree` accepts a single positive integer n as its argument and constructs a complete binary tree of depth n.
Input:

```
complete_binary_tree(2)
```

Output:

```
an undirected unweighted graph with 7 vertices and 6 edges
```

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

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

Output:

```
an undirected unweighted graph with 13 vertices and 12 edges
```

2.7 Creating graph from a graphic sequence: `is_graphic_sequence`, `sequence_graph`

The command `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 is based on Erdős–Gallai theorem and has the complexity $O(|L|^2)$.
Input:

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

Output:

```
true
```

The command `sequence_graph` accepts a list L of positive integers as its only argument. If the list represents a graphic sequence, the corresponding graph is constructed by using Havel–Hakimi algorithm with complexity $O(|L|^2 \log |L|)$. If the argument is not a graphic sequence, an error is returned.
Input:

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

Output:

```
an undirected unweighted graph with 8 vertices and 15 edges
```

2.8 Interval graphs: `interval_graph`

The command `interval_graph` takes as its argument a sequence or list of real-line intervals 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.
Input:

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

Output:

```
an undirected unweighted graph with 7 vertices and 10 edges
```

2.9 Star graphs: `star_graph`

The command `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.
Input:

```
star_graph(5)
```

Output:

```
an undirected unweighted graph with 6 vertices and 5 edges
```

2.10 Wheel graphs: `wheel_graph`

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

```
wheel_graph(5)
```

Output:

```
an undirected unweighted graph with 6 vertices and 10 edges
```

2.11 Web graphs: `web_graph`

The command `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)`.
Input:

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

Output:

```
an undirected unweighted graph with 21 vertices and 35 edges
```

2.12 Prism graphs: `prism_graph`

The command `prism_graph` accepts a positive integer n as its only argument and returns the prism graph with parameter n, namely `web_graph(n,2)`.
Input:

```
prism_graph(5)
```

Output:

```
an undirected unweighted graph with 10 vertices and 15 edges
```

2.13 Antiprism graphs: `antiprism_graph`

The command `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.
Input:

```
antiprism_graph(5)
```

Output:

```
an undirected unweighted graph with 10 vertices and 20 edges
```

2.14 Grid graphs: `grid_graph`, `torus_grid_graph`

The command `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)`.
Input:

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

Output:

```
an undirected unweighted graph with 15 vertices and 22 edges
```

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

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

Output:

```
an undirected unweighted graph with 15 vertices and 30 edges
```

2.15 Kneser graphs: `kneser_graph`, `odd_graph`

The command `kneser_graph` accepts two positive integers $n \leq 20$ and k as its arguments and returns the Kneser graph with parameters n, k. It 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. The Kneser graphs can get exceedingly complex even for relatively small values of n and k: note that the number of vertices is equal to $\binom{n}{k}$.
Input:

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

Output:

```
an undirected unweighted graph with 10 vertices and 15 edges
```

The command `odd_graph` accepts a positive integer $d \leq 8$ as its only argument and returns the Kneser graph with parameters $n = 2\,d + 1$ and $k = d$.
Input:

```
odd_graph(3)
```

Output:

```
an undirected unweighted graph with 10 vertices and 15 edges
```

The both examples above return the Petersen graph.

2.16 Sierpiński graphs: `sierpinski_graph`

The command `sierpinski_graph` accepts two or three arguments. Calling the command with two positive integers n and k produces the Sierpiński graph[1] S_k^n. If the symbol `triangle` is passed as the optional third argument, a Sierpiński triangle graph ST_k^n is returned. It is obtained by contracting all non-clique edges from S_k^n. In particular, ST_3^n is the well-known Sierpiński sieve graph of order n.
Input:

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

Output:

```
an undirected unweighted graph with 81 vertices and 120 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 42 vertices and 81 edges
```

2.17 Generalized Petersen graphs: `petersen_graph`

The command `petersen_graph` accepts one or two arguments: a positive integer n and optionally a positive integer k (which defaults to 2). The command returns generalized Petersen graph $P(n, k)$ which 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. If $k = 1$, the prism graph of order n is obtained.
For example, to obtain the dodecahedral graph $P(10, 2)$ one may input:

```
petersen_graph(10)
```

Output:

[1] For the complete definition and properties see the paper "A survey and classification of Sierpiński-type graphs" by Andreas M. Hinz et al.

```
an undirected unweighted graph with 20 vertices and 30 edges
```

To obtain Möbius–Kantor graph, input:

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

Output:

```
an undirected unweighted graph with 16 vertices and 24 edges
```

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

2.18 Creating isomorphic graphs: `isomorphic_copy`, `permute_vertices`, `relabel_vertices`

The three commands presented in this section are used to obtain isomorphic copies of an existing 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.
Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 4 edges
```

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.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 3 edges
```

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.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 3 edges
```

2.19 Extracting subgraphs of a graph: `subgraph`, `induced_subgraph`

The command `subgraph` accepts two arguments, a graph $G(V, E)$ and a list of edges $L \subset E$, and returns the subgraph of G formed by edges from L.
Input:

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

Output:

```
an undirected unweighted graph with 4 vertices and 4 edges
```

The command `induced_subgraph` accepts two arguments, a graph $G(V, E)$ and a list of vertices $L \subset V$, and returns the subgraph of G formed by all edges in E which have endpoints in L.
Input:

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

Output:

```
an undirected unweighted graph with 6 vertices and 6 edges
```

2.20 Underlying graph: `underlying_graph`

The command `underlying_graph` accepts a graph G as its only argument and returns the underlying graph of G obtained by dropping the directions of arcs and weights of edges/arcs.
Input:

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

Output:

```
a directed weighted graph with 3 vertices and 3 arcs
```

Input:

```
underlying_graph(G)
```

Output:

```
an undirected unweighted graph with 3 vertices and 3 edges
```

2.21 Reversing the edge directions: `reverse_graph`

The command `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. It 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.
Input:

```
G:=digraph(6, %{[1,2],[2,3],[2,4],[4,5]%}):;
      GT:=reverse_graph(G):; edges(GT)
```

Output:

```
Done, Done, [[2,1],[3,2],[4,2],[5,4]]
```

2.22 Graph complement: `graph_complement`

The command `graph_complement` accepts a graph $G(V, E)$ as its only argument and returns the complement $GC(V, EC)$ of G, where EC is the largest set containing only edges/arcs not present in G.
Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 5 edges
```

2.23 Union of graphs: `graph_union`, `disjoint_union`

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

```
G1:=graph([1,2,3],%{[1,2],[2,3]%}):;
G2:=graph([1,2,3],%{[3,1],[2,3]%}):; graph_union(G1,G2)
```

Output:

```
Done, Done, an undirected unweighted graph with 3 vertices and
3 edges
```

The command `disjoint_union` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k = 1, 2, \ldots, n$ as its argument and returns their disjoint union $G(V, E)$, 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 $|E| = \sum_{k=1}^{n} |E_k|$.
Input:

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

Output:

```
an undirected unweighted graph with 6 vertices and 5 edges
```

2.24 Joining two graphs: `graph_join`

The command `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.
Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 7 edges
```

2.25 Graph power: `graph_power`

The command `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 if and only if there exists a path of length at most k in G. The adjacency matrix A_k of G^k is obtained by adding powers of the adjacency matrix A of G together:

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

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

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

Output:

```
an undirected unweighted graph with 5 vertices and 7 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 5 vertices and 9 edges
```

2.26 Reversing the edge directions: `reverse_graph`

The command `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. It 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.
Input:

```
G:=digraph(6, %{[1,2],[2,3],[2,4],[4,5]%}):;
      GT:=reverse_graph(G):; edges(GT)
```

Output:

```
Done, Done, [[2,1],[3,2],[4,2],[5,4]]
```

2.27 Graph product: `cartesian_product`, `tensor_product`

The command `cartesian_product` accepts a sequence of graphs $G_k(V_k, E_k)$ for $k = 1, 2, \ldots, n$ 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 "v_1:v_2" where $v_1 \in V_1$ and $v_2 \in V_2$, such that ("u_1:v_1","u_2:v_2") is 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.
Input:

```
G1:=graph(trail(1,2,3,4,1,5)):; G2:=star_graph(3):;
```

Output:

```
Done, Done
```

Input:

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

Output:

```
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 $k = 1, 2, \ldots, n$ 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 "v_1:v_2" where $v_1 \in V_1$ and $v_2 \in V_2$, such that ("u_1:v_1","u_2:v_2") is in E if and only if u_1 is adjacent to u_2 **and** v_1 is adjacent to v_2.
Input:

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

Output:

```
an undirected unweighted graph with 20 vertices and 30 edges
```

2.28 Seidel switch: `seidel_switch`

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

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

Output:

```
an undirected unweighted graph with 5 vertices and 7 edges
```

3 Modifying graphs

3.1 Adding and removing vertices: `add_vertex`, `delete_vertex`

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.
Input:

```
add_vertex(complete_graph(5),6)
```

Output:

```
an undirected unweighted graph with 6 vertices and 10 edges
```

Input:

```
add_vertex(complete_graph(5),[a,b,c])
```

Output:

```
an undirected unweighted graph with 8 vertices and 10 edges
```

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

```
add_vertex(complete_graph([1,2,3,4,5]),[4,5,6])
```

Output (only vertex 6 is created):

```
an undirected unweighted graph with 6 vertices and 10 edges
```

Any vertex can be removed with the command `delete_vertex`. It accepts two arguments, a graph $G(V, E)$ and a single label v or a list of labels L, and returns the graph

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

or, if a list L is given,

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

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

```
delete_vertex(complete_graph(5),2)
```

Output:

```
an undirected unweighted graph with 4 vertices and 6 edges
```

Note that vertex removal implies deletion of incident edges.
Input:

```
delete_vertex(complete_graph(5),[2,3])
```

Output:

```
an undirected unweighted graph with 3 vertices and 3 edges
```

3.2 Adding and removing edges or arcs: `add_edge`, `add_arc`, `delete_edge`, `delete_arc`

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.
Input:

```
add_edge(cycle_graph(4),[1,3])
```

Output:

```
an undirected unweighted graph with 4 vertices and 5 edges
```

Input:

```
add_edge(cycle_graph(4),[1,3,5,7])
```

Output:

```
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.
Input:

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

Output:

```
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.
Input:

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

Output:

```
an undirected weighted graph with 4 vertices and 3 edges
```

3.3 Setting edge weights: `set_edge_weight`, `get_edge_weight`

The command `set_edge_weight` changes the weight of an edge/arc in a weighted graph. It accepts three arguments: a weighted graph $G(V, E)$, edge/arc $e \in E$ and the new weight w, which may be any number. The old weight is returned.

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.

For example, input:

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

Output:

```
an undirected weighted graph with 3 vertices and 2 edges
```

Input:

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

Output:

3.4 Contracting edges: `contract_edge`

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

```
contract_edge(complete_graph(5),[1,2])
```

Output:

```
an undirected unweighted graph with 4 vertices and 6 edges
```

To contract a set $\{e_1, e_2, \ldots, e_k\} \subset E$ 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)
```

Output:

```
an undirected unweighted graph with 5 vertices and 10 edges
```

Input:

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

Output:

```
an undirected unweighted graph with 3 vertices and 3 edges
```

3.5 Subdividing edges: `subdivide_edges`

The command `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.
Input:

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

Output:

```
an undirected unweighted graph with 7 vertices and 8 edges
```

Input:

```
subdivide_edges(complete_graph(2,3),[1,5],3)
```

Output:

```
an undirected unweighted graph with 8 vertices and 9 edges
```

3.6 Graph attributes: `set_graph_attribute`, `get_graph_attribute`, `discard_graph_attribute`, `list_graph_attributes`

The command `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 set by the user call the `discard_graph_attribute` command, which accepts two arguments: a graph G and a sequence or list of tags to be cleared, and returns the modified copy of G.

For example, input:

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

Output:

```
a directed unweighted graph with 3 vertices and 3 arcs
```

Input:

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

Output:

```
a directed unweighted graph with 3 vertices and 3 arcs
```

Input:

```
get_graph_attribute(G,"name")
```

Output:

```
"C3"
```

Input:

```
list_graph_attributes(G)
```

Output:

```
["directed"=true,"weighted"=false,"name"="C3",
                "shape"='triangle']
```

Input:

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

Output:

```
a directed unweighted graph with 3 vertices and 3 arcs
```

Input:

```
list_graph_attributes(G)
```

Output:

```
["directed"=true,"weighted"=false,"name"="C3"]
```

3.7 Vertex attributes: `set_vertex_attribute`, `get_vertex_attribute`, `discard_vertex_attribute`, `list_vertex_attributes`

The command `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.

3.8 Edge attributes: `set_edge_attribute`, `get_edge_attribute`, `discard_edge_attribute`, `list_edge_attributes`

The command `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.

4 Import and export

4.1 Loading graphs from dot files: `import_graph`

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 4.3.

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

```
G:=import_graph("dot/philosophers.dot"
```

Output:

```
an undirected unweighted graph with 21 vertices and 27 edges
```

4.2 Saving graphs to dot files: `export_graph`

The command `export_graph` accepts two arguments, a graph G and a string `filename`, and writes G to the file specified by `filename` using the dot language. `filename` 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. The name of the file should be specified at the end of the path, with or without `.dot` extension. The command returns 1 on success and 0 on failure.
Input:

```
export_graph(G,"dot/copy_of_philosophers")
```

Output:

```
1
```

4.3 The dot file format overview

Giac has the basic support for the dot language[2]. Each file is used to hold exactly one graph and should look like this:

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

```
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 this package 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.

- Creating isolated vertices: `vertex_name [attributes]?`
- Creating edges and trails: `V1 <edgeop> V2 <edgeop> ... <edgeop> Vk [attributes]?`
- Setting graph attributes: `graph [atributes]`

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 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};
}
```

5 Graph properties

6 Traversing graphs

6.1 Shortest path in unweighted graphs: `shortest_path`

The command `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 $O(|V|+|E|)$.

For example, input:

```
shortest_path(graph("dodecahedron"),1,9)
```

Output:

```
[1,5,4,9]
```

Input:

```
shortest_path(graph("dodecahedron"),1,[7,9])
```

Output:

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

6.2 Shortest path in weighted graphs: `dijkstra`

The command `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 edges along the path, which is required to be minimal.

6.3 Spanning trees: `spanning_tree`, `minimal_spanning_tree`

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.

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.

7 Visualizing graphs

7.1 Drawing graphs using various algorithms: `draw_graph`

The command `draw_graph` accepts one or two arguments, the mandatory first one being a 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 Tutte's barycentric method
- `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.

Graph drawings in the plane, when force-directed methods are used (i.e. when specifying `spring` or `planar` style), will often reflect axial symmetry if present in the graph. To make it more prominent in a drawing, rotation of the layout is performed in order for the corresponding axis to become vertical. This is done separately for every connected component of G.

Tree and circle 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).

7.2 Setting custom vertex positions: `set_vertex_positions`, `get_vertex_positions`

7.3 Highlighting parts of a graph: `highlight_vertex`, `highlight_edges`, `highlight_trail`, `highlight_subgraph`