Graph theory package for Giac/Xcas

Luka Marohnić

May 25, 2018

Contents

1 Introduction **2**

2 Constructing graphs **2**

2.1 Creating graphs from scratch : `graph`, `digraph` 2
2.2 Promoting to directed and/or weighted graphs : `make_directed`, `make_weighted` 8
2.3 Cycle graphs : `cycle_graph` 8
2.4 Path graphs : `path_graph` 8
2.5 Trail of edges : `trail` 8
2.6 Complete graphs : `complete_graph`, `complete_binary_tree`, `complete_kary_tree` 8
2.7 Creating graph from a graphic sequence : `is_graphic_sequence`, `sequence_graph` 8
2.8 Interval graphs : `interval_graph` 8
2.9 Star graphs : `star_graph` 8
2.10 Wheel graphs : `wheel_graph` 8
2.11 Web graphs : `web_graph` 8
2.12 Prism graphs : `prism_graph` 8
2.13 Antiprism graphs : `antiprism_graph` 8
2.14 Grid graphs : `grid_graph`, `torus_grid_graphs` 8
2.15 Kneser graphs : `kneser_graph`, `odd_graph` 8
2.16 Sierpinski graphs : `sierpinski_graph` 8
2.17 Generalized Petersen graphs : `petersen_graph` 8
2.18 Isomorphic copy of a graph : `isomorphic_copy`, `permute_vertices`, `relabel_vertices` 8
2.19 Extracting subgraphs of a graph : `subgraph`, `induced_subgraph` 8
2.20 Graph complement : `graph_complement` 8
2.21 Union of graphs : `graph_union`, `disjoint_union` 8
2.22 Joining two graphs : `graph_join` 8
2.23 Graph power : `graph_power` 8
2.24 Underlying graph : `underlying_graph` 8
2.25 Reversing arc directions in a digraph : `reverse_graph` 8
2.26 Graph product : `cartesian_product`, `tensor_product` . 8
2.27 Seidel switch : `seidel_switch` 8

3 Modifying graphs **8**

4 Import and export **8**

5 Graph properties **8**

6 Traversing graphs **8**

7 Visualizing graphs **8**

1 Introduction

This document contains an overview of the graph theory commands built in the Giac/Xcas software.

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 atomary 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`
- Dodecahedron 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`
- Icosahedron graph : `icosahedron`
- Levi graph : `levi`
- Ljubljana graph : `ljubljana`
- McGee graph : `mcgee`
- Möbius–Kantor graph : `mobius-kantor`
- Nauru graph : `nauru`
- Octahedron graph : `octahedron`
- Pappus graph : `pappus`
- Petersen graph : `petersen`
- Robertson graph : `robertson`
- Soccer ball graph : `soccerball`
- Tetrahedron 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.
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
```

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

2.3 **Cycle graphs :** cycle_graph

2.4 **Path graphs :** path_graph

2.5 **Trail of edges :** trail

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

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

2.8 **Interval graphs :** interval_graph

2.9 **Star graphs :** star_graph

2.10 **Wheel graphs :** wheel_graph

2.11 **Web graphs :** web_graph

2.12 **Prism graphs :** prism_graph

2.13 **Antiprism graphs :** antiprism_graph

2.14 **Grid graphs :** grid_graph, torus_grid_graphs

2.15 **Kneser graphs :** kneser_graph, odd_graph

2.16 **Sierpinski graphs :** sierpinski_graph

2.17 **Generalized Petersen graphs :** petersen_graph

2.18 **Isomorphic copy of a graph :** isomorphic_copy, permute_vertices, relabel_vertices

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

2.20 **Graph complement :** graph_complement

2.21 **Union of graphs :** graph_union, disjoint_union

2.22 **Joining two graphs :** graph_join

2.23 **Graph power :** graph_power

2.24 **Underlying graph :** underlying_graph

2.25 **Reversing arc directions in a digraph :** reverse_graph

2.26 **Graph product :** cartesian_product, tensor_product

2.27 **Seidel switch :** seidel_switch

3 Modifying graphs

4 Import and export

5 Graph properties

6 Traversing graphs