GRAPHS package for Maxima

Andrej Vodopivec
andrej.vodopivec@gmail.com

Introduction

The `GRAPHS` package provides graph and digraph data structure for Maxima. Graphs and digraphs are simple (have no multiple edges nor loops), although digraphs can have a directed edge from *u* to *v* and a directed edge from *v* to *u*.

This document gives basic documentation about implementation of data structures and lists all implemented functions.

There are some examples at the end of this paper (some examples are screen images taken from wxMaxima, the output will look a little different in other interfaces).

Installation

The `GRAPHS` package has been tested with Maxima 5.12. It should also work with Maxima 5.11 but you should also install the `draw` package.

Unpack the `graph.zip` archive somewhere in you Maxima path and then load the package into Maxima with `load("graphs/graphs").`

Data structures

A graph is an incidence structure *G=(V,E)* where *V* is a finite set of vertices and *E* set of edges. For undirected graphs, an edge *e* is a 2-set *{u, v}* of vertices *u, v* from *V*. For a directed graph, an edge is an ordered pair *(u, v)* of vertices *u, v* from *V*. We call a directed edge an arc.

Undirected graphs are represented by a `graph` data structure and directed graphs are represented by a `digraph`. Internally graphs are represented by adjacency lists, implemented as lisp structures. Vertices are identified by their ids (an id is an integer). Labels can be assigned to vertices of graphs/digraphs and weights can be assigned to edges/arcs of graphs/digraphs.

Creating graphs and digraphs

The first set of functions create special classes of graphs:

- `dodecahedron_graph();`
- `wheel_graph(n);`
- `circulant_graph(n, d);`
- `petersen_graph()`: returns the Petersen graph;
- `petersen_graph(n, d)`: returns the generalized Petersen graph $P_{n,d}$;
- `cycle_graph(n)`: returns a cycle on *n* vertices;
- `cycle_digraph(n)`: returns a directed cycle on *n* vertices;
- `path_graph(n)`: returns a path on *n* vertices;
- `path_digraph(n)`: returns a directed path on *n* vertices;
- `complete_graph(n)`: returns a complete graph on *n* vertices;

- `empty_graph(n)`: returns an empty graph on *n* vertices;
- `new_graph()`: returns a new graph with no vertices or edges;
- `random_graph(n, p)`: returns a random graph on *n* vertices – each edge is present with probability *p;*
- `random_graph1(n,m)`: returns a random graph on *n* vertices with random *m* edges;
- `random_regular_graph(n, [d])`: returns a random *d*-regular graph on *n* vertices; if *n* is not given, it returns a random 3-regular graph (if *d* is odd, *n* should be even),
- `random_digraph(n, p)`: returns a random directed graph on *n* vertices – each arc is present with probability *p;*
- `random_tournament(n)`: returns a random tournament on *n* vertices;
- `random_network(n, p, w)`: returns a random network on *n* vertices; each arc is present with probability *p*; each present arc has a weight in the range *[0, w]*;
- `random_tree(n)`: returns a random tree on *n* vertices.

The second set of functions create graphs form other graphs or structures:

- `graph_product(g1, g2)`: returns the cross product of graphs *g1* and *g2*;
- `line_graph(g)`: returns the line graph of graph *g*;
- `complement_graph(g)`: returns the complement of graph *g*;
- `graph_union(g1, g2)`: returns the union of graph *g1* and *g2*;
- `underlying_graph(g)`: returns the underlying graph of a directed graph *g*;
- `from_adjacency_matrix(A)`: returns the graph represented by adjacency matrix *A*;
- `induced_subgraph(V, g)`: returns the subgraph of *g* induced on the subset *V* of vertices of graph *g*;
- `copy_graph(g)`: returns a copy of the graph *g*;
- `create_graph(v_list, e_list, [dir])`: creates a new graph; *v_list* can be a list of vertices $[v_1,...,v_n]$ or a list of vertices with labels $[[v_1, l_1], ..., [v_n,l_n]]$ or an integer *n* and the vertices are then taken as $[0,1,...,n]$; *e_list* is a list of edges $[e_1,...,e_m]$ or a list of edges with weights $[[e_1, w_1], ...,[e_m,w_m]]$; *dir* is an optional argument – if it is not `false` the graph will be directed.

Graph properties

This section describes functions for examining graph properties. When a property makes sense for both graphs and digraphs, the same function is used for both, but some functions can only be used for graphs or only for digraphs.

Functions for examining properties of both graphs and digraphs:

- `is_edge_in_graph(e, g)`: tests if the edge *e* is in the graph *g*;
- `is_vertex_in_graph(v, g)`: tests if the vertex v is in the graph g;
- `is_graph(g);`
- `is_digraph(g);`
- `is_graph_or_digraph(g);`

- `graph_size(g)`: returns the number of vertices of the graph *g*;
- `graph_order(g)`: returns the number of edges/arcs of the graph *g;*
- `vertices(g)`: returns the list of vertices of the graph *g*;
- `edges(g)`: returns the list of edges/arcs of the graph *g*;
- `set_vertex_label(v, l, g)`: sets the label of vertex *v* in *g* to *l*;
- `get_vertex_label(v, g)`: returns the label of the vertex *v* in *g* or `false` if *v* has no label;
- `clear_vertex_label(v, g)`: clears the label of vertex *v* in *g*;
- `set_edge_weight(e, w, g)`: sets the weight of edge/arc *e* in *g* to *w*;
- `get_edge_weight(e, g, [ifnot])`: returns the weight of edge/arc *e* in *g* of `false` if *e* has no weight; the default weight of edges is 1; if the edge is not in the graph, it produces an error or returns the optional argument `ifnot`.
- `clear_edge_weight(e, g)`: clears the weight of edge/arc *e* in *g*;
- `vertex_distance(u, v, g)`: returns the distance between *u* and *v* in *g*;
- `shortest_path(u, v, g)`: returns a list of consecutive vertices in the shortest path from *u* to *v* in *g*;
- `hamilton_cycle(g`*):* returns a Hamilton cycle in the graph *g* or an empty list if no Hamilton cycles exist;
- `hamilton_path(g`*):* returns a Hamilton path in the graph *g* or an empty list if no Hamilton paths exist.

Functions for examining properties of graphs:

- `is_connected(g)`: returns `true` if *g* is a connected graph and `false` otherwise;
- `connected_components(g)`: returns a list the vertex sets of connected components of the graph *g*;
- `is_biconnected(g)`: returns `true` if *g* is a 2-connected graph and `false` otherwise;
- `biconnected_components(g)`: returns a list of the vertex sets of 2-connected components of *g*;
- `adjacency_matrix(g)`: returns the adjacency matrix of graph *g*;
- `laplacian_matrix(g)`: returns the Laplacian matrix of graph *g*;
- `graph_charpoly(g, x)`: returns the characteristic polynomial of the adjacency matrix of the graph *g*;
- `graph_eigenvalues(g)`: returns the eigenvalues of graph *g* – the output has the same format as maxima `eigenvalues` function;
- `max_clique(g)`: returns a maximum clique in the graph *g*;
- `max_independent_set(g)`: returns a maximum independent set in the graph *g*;
- `neighbors(v, g)`: returns the list of neighbors of vertex *v* in graph *g*;
- `vertex_degree(v, g)`: returns the degree of vertex *v* in graph *g*;
- `degree_sequence(g)`: returns a list of degrees of vertices of *g*;

- `min_degree(g)`: returns a list *[d, v]*, where *v* is a vertex of minimum degree *d* in *g*;
- `max_degree(g)`: returns a list *[d, v]*, where v is a vertex of maximum degree *d* in *g*.
- `average_degree(g)`: returns the average degree of the graph *g*;
- `bipartition(g)`: returns a list *[A, B]*, where *A* and *B* are bipartition of vertices of *g*;
- `is_bipartite(g)`: returns `true` if *g* is bipartite and `false` otherwise;
- `girth(g)`: returns the length of the shortest cycle in *g*;
- `odd_girth(g)`: returns the length of the shortest odd cycle in *g*;
- `diameter(g)`: returns the diameter of *g*;
- `radius(g)`: returns the radius of *g*;
- `is_tree(g)`: returns `true` if *g* is a tree and `false` otherwise;
- `minimum_spanning_tree(g)`: returns the minimum spanning tree of a weighted graph *g* – if there is no weight on an edge, weight 1 is assumed;
- `vertex_coloring(g)`: returns the optimal coloring of vertices of the graph *g*; return value is *[chromatic_number, [[v_1, c_1],...,[v_n, c_n]]]* where chromatic_number is the number of different numbers and c_i is the color of the vertex v_i in an optimal coloring; `vertex_coloring` uses a backtracking algorithm to color the graph;
- `chromatic_number(g)`: return the chromatic number of graph *g*;
- `edge_coloring(g)`: returns an optimal coloring of edges of the graph *g*; return value is *[chromatic_index, [[e_1,c_1],...,[e_m,c_m]]],* where *chromatic_index* is the number of colors in an optimal coloring and c_i is the color of edge e_i; `edge_coloring` uses `vertex_coloring` on the line graph of *g*;
- `chromatic_index(g)`: returns the chromatic index of the graph *g*.

Functions for examining properties of digraphs:

- `strong_components(g)`: returns the strong components of a digraph g,
- `is_sconnected(g)`: returns true is digraph g is strongly connected,
- `in_neighbors(v, g)`: returns the list of in-neighbors of vertex *v* in digraph *g*;
- `out_neighbors(v, g)`: returns the list of in-neighbors of vertex *v* in digraph *g*;
- `vertex_in_degree(v, g)`: returns the in-degree of vertex *v* in digraph *g*;
- `vertex_out_degree(v, g)`: returns the out-degree of vertex *v* in digraph *g*;
- `max_flow(net, source, sink)`*:* returns a maximum flow through network *net* from *source* to *sink*; return value is *[val, [[e_1, fl_1], [e_2, fl_2], ..., [e_m, fl_m]]]*, where *val* is the value of the flow and fl_i is the value of the flow on edge e_i.

Functions for modifying graphs

Note that these functions modify the input graph and return `done` on success.

- `add_vertex(v, g)`: adds a new vertex *v* to (di)graph *g*;
- `add_vertices(vl, g)`: adds vertices from a list *vl* to (di)graph *g*;
- `add_edge(e, g)`: adds a new edge *e* to (di)graph *g*;

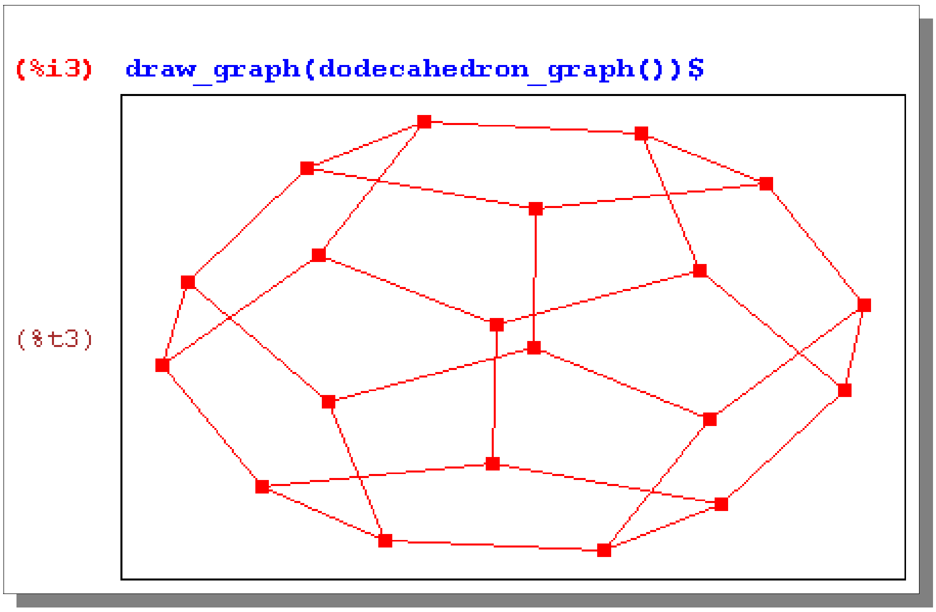
- `add_edges(el, g)`: adds edges from a list *el* to (di)graph *g*;
- `connect_vertices(v1, v2, g)`: add edges between *v1* and *v2*; *v1* and *v2* can be lists or vertices or a single vertex; if *v1* and *v2* are lists, all edges between *v1* and *v2* are added;
- `remove_vertex(v, g)`: removes the vertex *v* from (di)graph *g*;
- `remove_edge(e, g)`: removes the edge *e* from (di)graph *g*;
- `contract_edge(e, g)`: contracts the edge *e* in graph *g*.

Visualizing graphs

There is a function `draw_graph`, which is used to draw graphs. It can use graphviz programs to draw graphs nicely (graphviz programs are available from http://www.graphviz.org). It accepts some optional arguments:

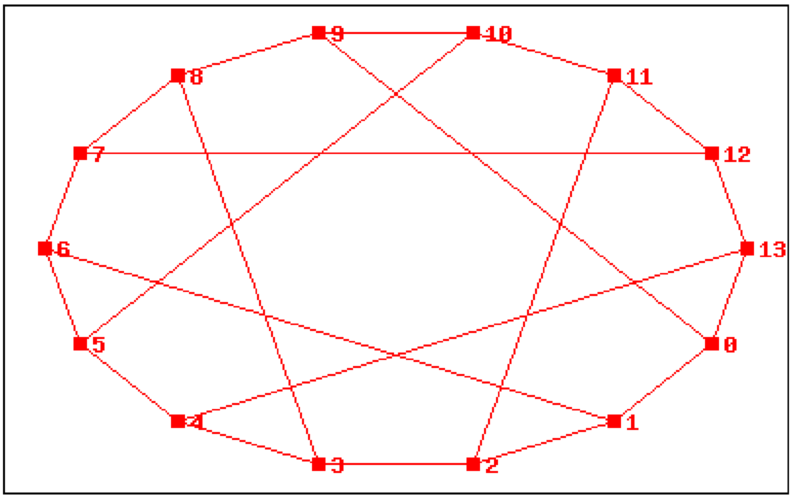
- `program`: one of graphviz programs (dot, neato, twopi, circ, fdp) if graphiviz is installed or circular; the default is neato, which implies that graphviz should be installed for `draw_graph` to work, if graphviz is not installed only the circular option is supported,
- `vertex_type`: see `point_type` in draw package,
- `show_id`: show the ids for vertices,
- `terminal`: gnuplot terminal (one of screen, wxmaxima, png, ps),
- `file_name`: if terminal is png or ps, the name of the file in which to plot.

Examples:



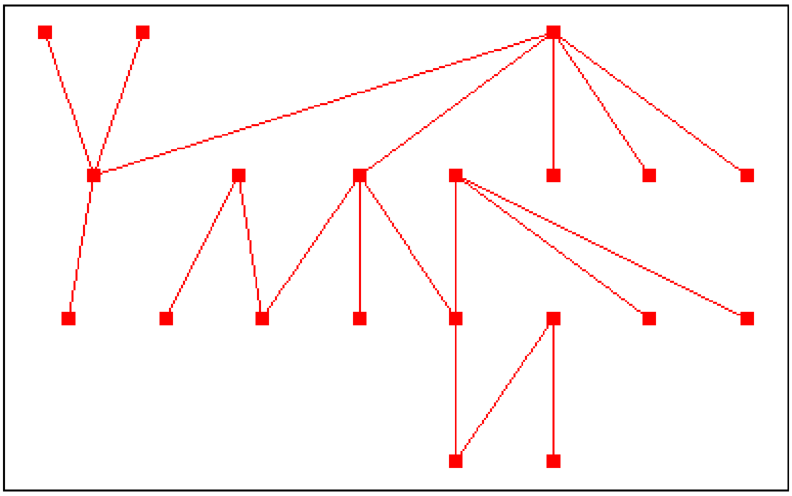
```
(%i7) draw_graph(
          heawood_graph(),
          program=circular,
          show_id=true
      )$
```

(%t7)




```
(%i7) draw_graph(random_tree(20), program=dot);
```

(%t7)



Data structures

The GRAPHS package exposes the underlying lisps datastructure *hashtable*. Functions for using hasthables:

- `hash_table()`: creates the hashtable
- `set_hash(elt, ht, val)`: enters the value *val* under the key *elt* into the hashtable *ht*;
- `get_hash(elt, ht, [ifnot])`: gets the value under the key *elt* in the hashtable *ht*; if the key *elt* is not in *ht* then it returns `false` or the optional argument `ifnot`.
- `hash_table_data(ht)`: returns the data stored in the hashtable *ht* as a list *[key1->val*$_1$*, key*$_2$*->val*$_2$*,...,key*$_n$*->val*$_n$*]*.

Example:

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) ht : hash_table()$
(%i3) set_hash(1, ht, 4.1)$
(%i4) get_hash(1, ht);
(%o4) 4.1
(%i5) get_hash(2, ht);
(%o5) false
(%i6) get_hash(2, ht, inf);
(%o6) inf
(%i8) set_hash(2, ht, [1,2,3]);
(%o8) [1,2,3]
(%i9) hash_table_data(ht);
(%o9) [2->[1,2,3],1->4.1]
```

Example – the Petersen graph

In this section we investigate the properties of the Petersen graph.

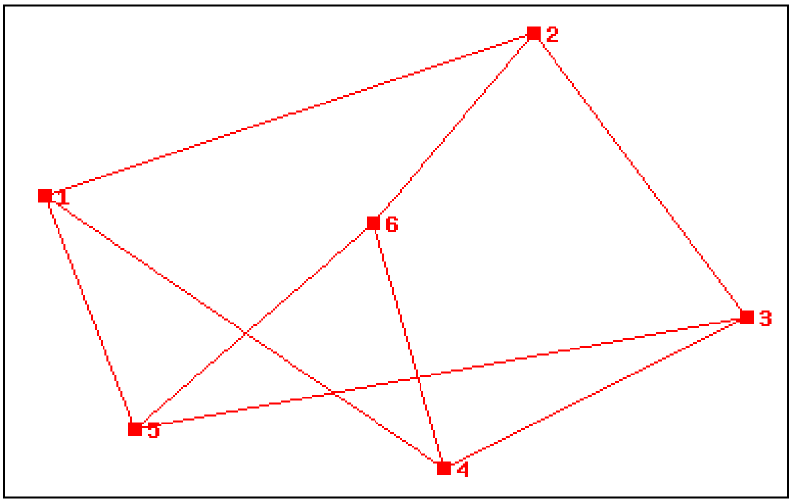
```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) p: petersen_graph();
(%o2) Structure [GRAPH]
(%i3) vertices(p);
(%o3) [9,8,7,6,5,4,3,2,1,0]
(%i4) edges(p);
(%o4) [[0,4],[6,9],[4,9],[3,4],[5,8],[3,8],[2,3],[7,9],[2,7],
[1,2],[6,8],[1,6],[0,1],[5,7],[0,5]]
(%i5) print_graph(p);
Graph on 10 vertices with 15 edges.
Adjacencies:
  9 :  6  4  7
  8 :  5  3  6
  7 :  9  2  5
  6 :  9  8  1
  5 :  8  7  0
  4 :  0  9  3
  3 :  4  8  2
  2 :  3  7  1
  1 :  2  6  0
  0 :  4  1  5
(%i6) neighbors(0, p);
(%o6) [4,1,5]
(%i7) girth(p);
(%o7) 5
(%i8) chromatic_index(p);
(%o8) 4
(%i9) max_independent_set(p);
(%o9) [0,2,8,9]
(%i10) hamilton_cycle(p);
(%o10) []
(%i11) hamilton_path(p);
(%o11) [0,5,7,2,1,6,8,3,4,9]
(%i12) vertex_distance(1, 5, p);
(%o12) 2
(%i13) shortest_path(1, 5, p);
(%o13) [1,0,5]
(%i14) graph_eigenvalues(p);
(%o14) [[3,-2,1],[1,4,5]]
(%i15) factor( graph_charpoly(p, x) );
(%o15) (x-3)*(x-1)^5*(x+2)^4
```

Examples – **create_graph** *function*

Example for the `create_graph` function:

```
(%i1)  load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.

(%i2)  G : create_graph(
           [1, 2, 3, 4, 5, 6],
           [
            [1,2], [2,3], [3,4], [1,4], [1,5],
            [5,3], [2,6], [6,4], [5, 6]
           ]
       )$

(%i3)  draw_graph(G, show_id=true)$
```

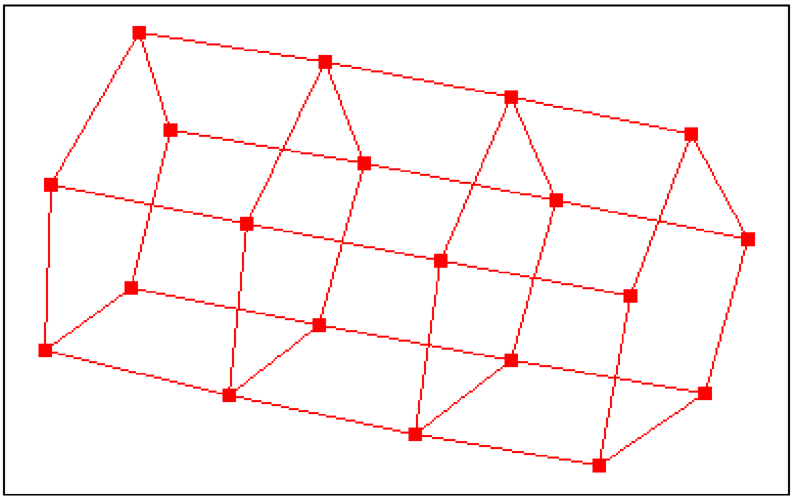
(%t3)



Example – creating graphs

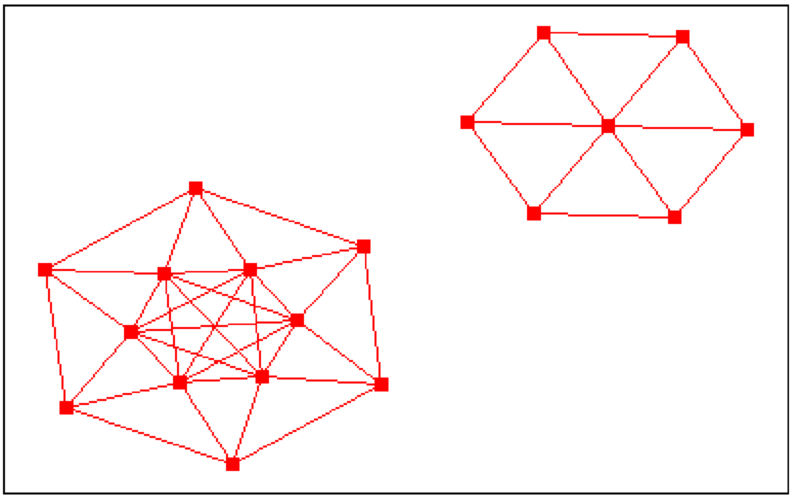
Draw the graph $C_5 \times P_4$

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.

(%i2) g: graph_product(cycle_graph(5), path_graph(4));
(%o2) Structure [GRAPH]

(%i3) draw_graph(g)$
(%t3)
```



Draw the graph W_6 and its line graph.

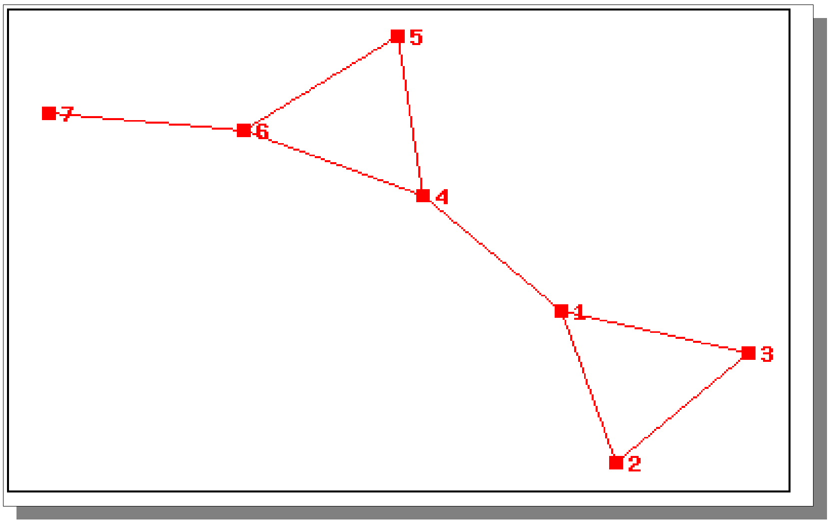
```
(%i4) g : wheel_graph(6)$
      h : line_graph(g)$

(%i6) draw_graph(graph_union(g, h));
(%t6)
```



Example – connectivity

We investigate the connectivity of the graph on the image:

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) g : create_graph(
    [1,2,3,4,5,6,7],
    [
     [1,2], [2,3], [1,3],
     [4,5], [5,6], [4,6],
     [1,4], [6,7]
    ]
)$
(%i3) is_connected(g);
(%o3) true
(%i4) is_biconnected(g);
(%o4) false
(%i5) biconnected_components(g);
(%o5) [[6,7],[4,5,6],[1,4],[1,2,3]]
```

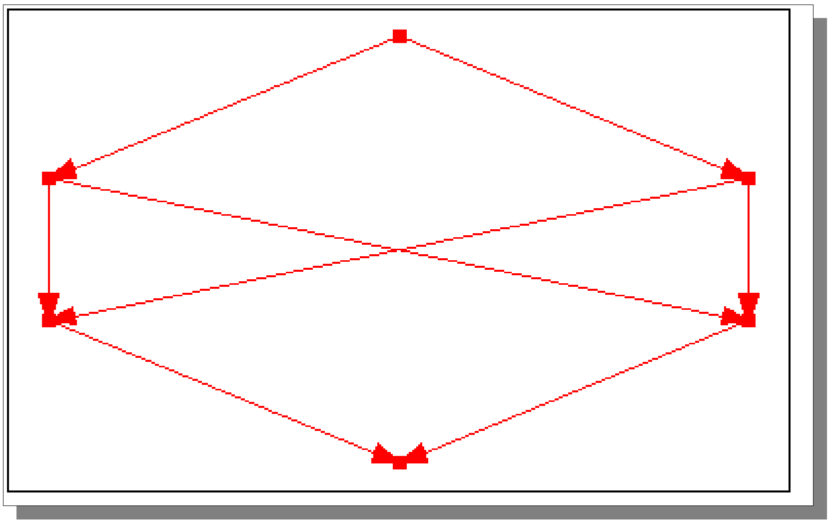


Example – max-flow problem

Here we create a network and compute the max-flow over it.

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i5) net : create_graph(
    [1,2,3,4,5,6],      /* the list of vertices in the network */
    [
     [[1,2], 1.2],      /* the list of arcs with weights */
     [[1,3], 2.1],
     [[2,4], 2.3],
     [[2,5], 1.2],
     [[3,4], 4,1],
     [[3,5], 1.1],
     [[5,6], 0.5],
     [[4,6], 1.2]
    ],
    'directed          /* network is a directed graph */
);
(%o5) Structure [DIGRAPH]
(%i6) max_flow(net, 1, 6);
(%o6) [1.7,[[[1,2],1.2],[[1,3],0.5],[[2,4],1.2],[[2,5],0],
[[3,4],0],[[3,5],0.5],[[5,6],0.5],[[4,6],1.2]]]
```

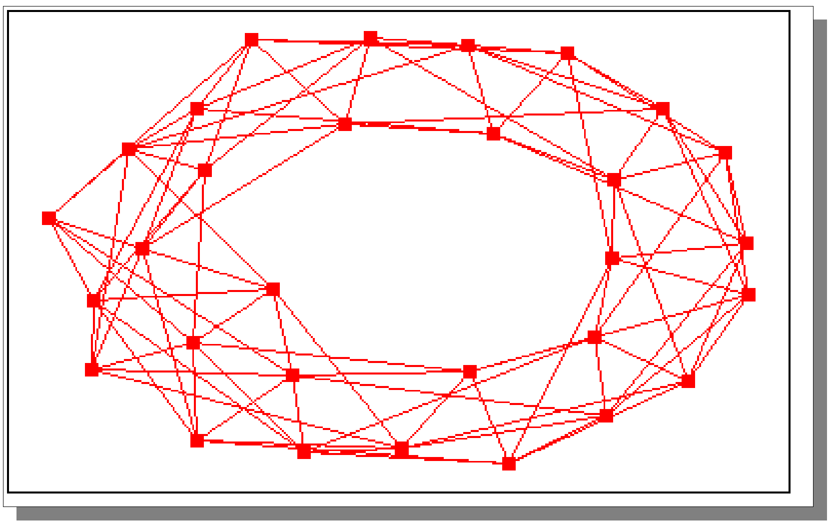
Figure of the network with the dot program:



Example – graphs from matrices

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) M : genmatrix(
         lambda([i,j], if remainder(abs(i-j), 5)=2
                       then 1
                       else 0),
         30, 30
      )$
(%i3) g : from_adjacency_matrix(M)$
(%i4) girth(g);
(%o4) 4
(%i5) odd_girth(g);
(%o5) 7
(%i6) chromatic_number(g);
(%o6) 3
(%i7) chromatic_index(g);
(%o7) 7
(%i8) max_degree(g);
(%o8) [7,27]
(%i9) vertex_degree(27, g);
(%o9) 7
(%i10) max_independent_set(g);
(%o10) [0,4,5,9,10,14,15,19,20,24,25,28,29]
(%i11) radius(g);
(%o11) 4
(%i12) vrt : sublist(vertices(g), evenp)$
(%i13) h : induced_subgraph(vrt, g)$
(%i14) degree_sequence(h);
(%o14) [3,3,3,3,3,3,4,4,4,4,4,4,4,4,4]
```

The graph *g*:



Example – BFS tree

Visit vertices of the tree along a BFS tree

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) bfs(g) := block(
    [Visited, Active, u, v, prev],
    /* a list of active vertices - initialy add the first vertex
       to the list of active vertices */
    Active : [first(vertices(g))],
    /* a list of already visited vertices */
    Visited : [],
    /* prev stores the back edges */
    prev : hash_table(),
    /* repeat while we still have active vertices */
    while length(Active) > 0 do (
        /* visit the first active vertex */
        u : first(Active),
        Active : rest(Active),
        Visited : cons(u, Visited),

        /* add the vertex to the tree */
        if get_hash(u, prev) # false then
            Tree : cons([get_hash(u, prev), u], Tree)
        else
            Tree : cons(u, Tree),

        /* check all neighbors of u - if there is some neighbor
           which has not been visited and is not active, add it
           to the list of active vertices */
        for v in neighbors(u, g) do (
            if not(member(v, Visited)) and
               not(member(v, Active))
            then (
                set_hash(v, prev, u),
                Active : endcons(v, Active)
            )
        )
    ),
    reverse(Tree)
)$
(%i3) p : petersen_graph()$
(%i4) bfs(p);
/* the edges of a BFS tree with the root 9 */
(%o4) [9,[9,6],[9,4],[9,7],[6,8],[6,1],[4,0],[4,3],[7,2],[7,5]]
```

Example – minimum cost spanning tree

In this section we implement an $O(n^2)$ algorithm for computing the minimum cost spanning tree.

```
(%i1) load("graphs/graphs")$
Loading graph theory package 'graphs', version 1.0.
(%i2) mcsp(g) := block(
    [V : vertices(g), U, v, u, n : graph_size(g), UmV, close,
     Tree : []],
 /* We grow the tree from an arbitrary vertex and in each step we
    add one vertex to the tree. U is the set of vertices in the
    tree, UmV is the set of vertices not in the Tree and tree is
    the set of edges in the tree */
    U : [V[1]],
    UmV : delete(U[1], V),
    /* In the hash table close we have for each vertex v the
       vertex closest to v in U */
    close : hash_table(),
    for v in UmV do set_hash(v, close, U[1]),
    /* Repeat until all vertices of g are in the tree */
    while length(U) < n do block(
        [closest, dist : inf],
        /* Choose a vertex in UmV closest to U */
        for v in UmV do (
            if get_edge_weight([v, get_hash(v, close)], g, inf) <
               dist
            then (
                dist:get_edge_weight([v, get_hash(v, close)], g),
                closest : v
            )
        ),
        /* Add this vertex to the tree */
        U : cons(closest, U),
        UmV : delete(closest, UmV),
        Tree : cons([closest, get_hash(closest, close)], Tree),
        /* Update the hash table close */
        for v in UmV do (
            if get_edge_weight([v, closest], g, inf) <
               get_edge_weight([v, get_hash(v, close)], g, inf)
            then
                set_hash(v, close, closest)
        )
    ),
    Tree
)$
(%i3) p : petersen_graph();
(%o3) Structure [GRAPH]
(%i4) for e in edges(p) do set_edge_weight(e, random(100), p);
(%o4) done
(%i5) mcsp(p);
(%o5) [[3,2],[2,1],[1,0],[7,5],[5,8],[8,6],[0,4],[4,9],[6,9]]
```