Introduction to Maple's GraphTheory Package

Michael Monagan

Department of Mathematics

Simon Fraser University

mmonagan@cecm.sfu.ca

Introduction

Maple's **GraphTheory** package was developed by a group of graduate students and faculty at Simon Fraser University under the direction of Michael Monagan starting in 2004. The design of the package was first presented at the 2005 Maple conference in Waterloo in the summer of 2005. New commands and improvements, in particular to facilities for drawing graphs, were presented at the 2006 Maple conference. The first version of the package was released in Maple 11 in 2007 as the **GraphTheory** package. Since then further improvements have been added and more improvements will appear in Maple 17 this year. The package supports simple undirected graphs and simple directed graphs, both of which may be weighted. At this time there is no support for multi-graphs.

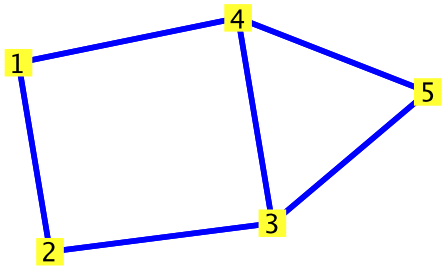
It has been a very stimulating experience for me as director of the project. Graph theory is such a rich area with an inexhaustible list of improvements that one might wish to add and a wide variety of software development issues that needed to address. We've been fortunate to have many dedicated and very talented graduate students work on the package. The package itself has a bit of everything. It has basic tools for creating and manipulating graphs, a subpackage of known special graphs from the literature, tools for creating random and non-isomorphic graphs, for exporting and importing graphs, animations of algorithms for teaching, programs for computing graph properties and graph polynomials, some of which are known to be NP-complete and NP-hard, and lastly, but perhaps most useful, tools for drawing graphs. This article gives you a taste. More information within Maple can be found at the main help page ?GraphTheory and the main example help page ?examples,GraphTheory.

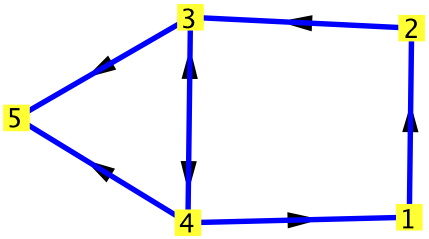
A quick tour

To use the GraphTheory package we first load it.

```
> with(GraphTheory):
```

Here are two graphs with 5 vertices. The first is a simple undirected graph. The second is a simple directed graph.





The two graphs may be input in Maple using the **Graph** command as shown below. The difference is that for an undirected graph we input edges in set braces { } and for a directed graph we use square brackets [ ] .

```
> G := Graph(5, {{1,2},{2,3},{3,4},{4,1},{3,5},{4,5}});
```

G := Graph 1: an undirected unweighted graph with 5 vertices and 6 edge(s)

G := Graph 1: an undirected unweighted graph with 5 vertices and 6 edge(s)

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

H := Graph 2: a directed unweighted graph with 5 vertices and 7 arc(s)

Notice that the default output is a one-line description of the graph. The two drawings of the graphs above were created with the DrawGraph command as follows. I'll explain later the **style=spring** option which two of the students and I worked long hours on.

```
> DrawGraph(G,style=spring);
> DrawGraph(H,style=spring);
```

The package has many commands, because that's the nature of the subject. Commands are input like other Maple commands. There are commands for testing for properties. For example the graph G has no Eulerian tour but it is planar.

```
> IsEulerian(G);
```

false

```
> IsPlanar(G,'Faces');
```

true

```
> Faces;
```

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

The faces computed correspond to the drawing. [4,1,2,3] means the cycle [4,1,2,3,4],

which is the inner square; [4,3,5] is the triangle; and [3,2,1,4,5] is the outside face. There are commands for testing for structural information that allow you to program with graphs. Here is the list of neighbors of G, the arrivals and departures for H. Notice that there is no departure from vertex 5 in H.

```
> Neighbors(G);
```

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

```
> Arrivals(H);
```

[[4], [1], [2, 4], [3], [3, 4]]

```
> Departures(H);
```

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

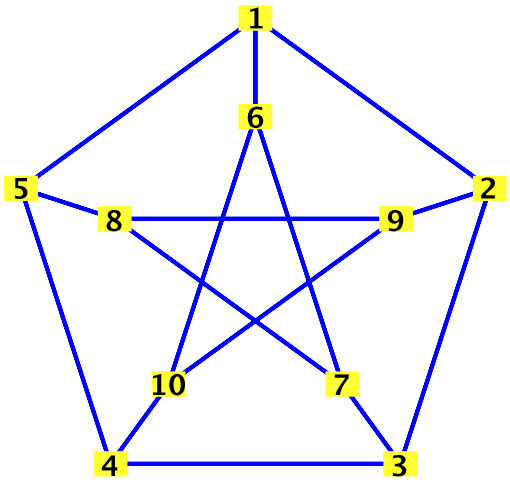
Special Graphs

A library of standard special graphs (and some not so standard) is available. For example, here is the well known Petersen graph.

```
> P := SpecialGraphs[PetersenGraph]();
```

P:= Graph 3: an undirected unweighted graph with 10 vertices and 15 edge(s)

```
> DrawGraph(P);
```



It is well known that the Petersen graph is not planar but it is 3-colorable.

```
> IsPlanar(P);
```

false

```
> IsVertexColorable(P,3,'C');
```

true

```
> C;
```

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

Each list in C is the list of vertices which have the same color. Here is how we can color the vertices to visualize the coloring.

```
> HighlightVertex(P,[1,3,8,10],red);
  HighlightVertex(P,[2,4,6],green);

> DrawGraph(P);
```

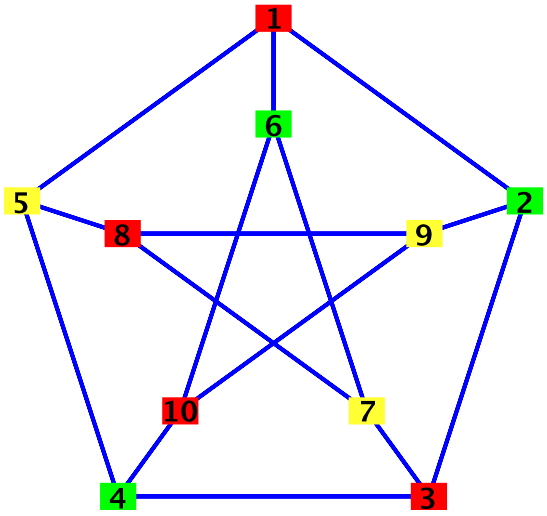


Table 1 lists Maple commands for computing polynomials related to graphs.

AcyclicPolynomial(G,p) CharacteristicPolynomial(G,x) ChromaticPolynomial(G,λ) FlowPolynomial(G,q) GraphPolynomial(G,x,y) RankPolynomial(G,x,y) SpanningPolynomial(G,p) ReliabilityPolynomial(G,p) - added for Maple 17 TuttePolynomial(G,x,y)
Table 1 Maple commands for computing polynomials related to graphs

Here is the Chromatic polynomial for the Petersen graph. It counts the number of ways a graph G can be colored with λ colors.

```
> CP := ChromaticPolynomial(P,lambda);
```

$$CP := \lambda\,(\lambda-1)\,(\lambda-2)\,\left(\lambda^7-12\,\lambda^6+67\,\lambda^5-230\,\lambda^4+529\,\lambda^3-814\,\lambda^2+775\,\lambda-352\right)$$

It is a polynomial in λ . You can see that $\lambda = 2$ is a root of the polynomial. That means there are 0 ways to color the graph with 2 colors.

```
> eval(CP,lambda=3);
```

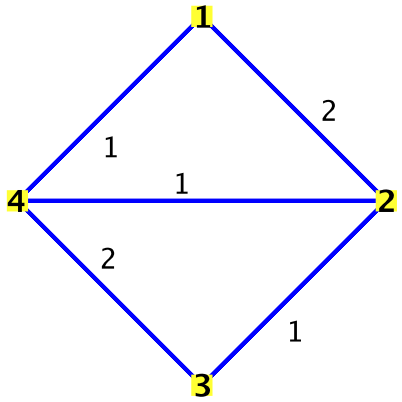
$$120$$

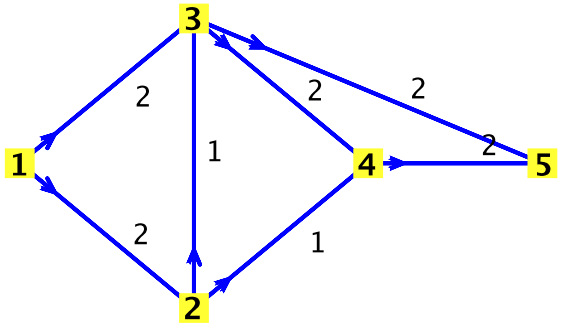
```
> eval(CP,lambda=2);
```

$$0$$

Weighted graphs and networks.

Here is an example of a weighted *undirected* graph and a weighted *directed* graph (a network).





The two graphs were input using the following commands.

```
> G := Graph(4,{[{1,2},2],[{2,3},1],[{3,4},2],[{4,1},1],[{2,4},1]});
```

G := Graph 4: an undirected weighted graph with 4 vertices and 5 edge(s)

```
> N := Graph(5,{[[1,2],2],[[1,3],2],[[2,3],1],[[2,4],1],[[3,5],2],[[4,
  5],2],[[3,4],2]});
```

N := Graph 5: a directed weighted graph with 5 vertices and 7 arc(s)

The two graphs were drawn with the following commands.

```
> DrawGraph(G);
> DrawNetwork(N,horizontal);
```

Here is a minimal spanning tree of G - it's the 3 edges with weight 1.

```
> T := MinimalSpanningTree(G);
```

T := Graph 6: an undirected weighted graph with 4 vertices and 3 edge(s)

```
> Edges(T,weights);
```

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

And here is the maximum flow for the network N which is 4 and a flow matrix achieving that flow.

```
> MaxFlow(N,1,5);
```

$$4, \begin{bmatrix} 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Well, you can see that there are a lot of commands. Many of the commands are accessible from the context menu for a graph. Try right-clicking on one of the graph objects e.g. Graph 6 in **(2.18)** above.

Graph input and export

```
> restart;
  with(GraphTheory):
```

The Graph command is the main command for creating graphs. It takes 4 main arguments, which may be given in any order.

n - the number of vertices
V - a list of vertex labels (can be integers, symbols or strings)
D - either directed or undirected
E - edge information which can be any of
- set of (weighted) edges,
- array of neighbor sets,
- trails or
- the adjacency matrix.

The Graph command is smart in that you don't need to specify *n*, *V* or *D* if they can be deduced from the edge information.

Here are the four ways to input the cycle *(a,b,c,d)*.

```
> G := Graph({{a,b},{b,c},{c,d},{d,a}});
```

G := Graph 1: an undirected unweighted graph with 4 vertices and 4 edge(s)

```
> G := Graph(4,[a,b,c,d],Array(1..4,[{2,4},{1,3},{2,4},{3,1}]));
```

G := Graph 2: an undirected unweighted graph with 4 vertices and 4 edge(s)

```
> G := Graph(Trail(a,b,c,d,a));
```

G := Graph 3: an undirected unweighted graph with 4 vertices and 4 edge(s)

```
> A := Matrix([[0,1,0,1],[1,0,1,0],[0,1,0,1],[1,0,1,0]]);
```

$$A := \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

```
> G := Graph([a,b,c,d],A);
```

G := Graph 4: an undirected unweighted graph with 4 vertices and 4 edge(s)

You can also create the graph one or more edges at a time.

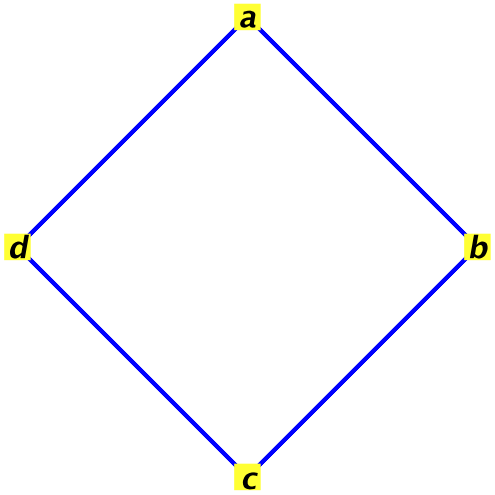
```
> G := Graph([a,b,c,d]):
```

```
> AddEdge(G,{a,b});
```
Graph 5: an undirected unweighted graph with 4 vertices and 1 edge(s)

```
> AddEdge(G,{b,c});
```
Graph 5: an undirected unweighted graph with 4 vertices and 2 edge(s)

```
> AddEdge(G,{{c,d},{d,a}});
```
Graph 5: an undirected unweighted graph with 4 vertices and 4 edge(s)

```
> DrawGraph(G,style=circle);
```



Maple uses a list-of-neighbors representation for storing the graph. It stores the edges as an array of sets of vertices. You can see Maple's data representation for a graph by using the lprint command.

```
> lprint(G);
```
```
GRAPHLN(undirected, unweighted, [a, b, c, d], Array(1 .. 4, {1 = {2, 4},
2 = {1, 3}, 3 = {2, 4}, 4 = {1, 3}}, datatype = anything, storage =
rectangular, order = Fortran_order), `GRAPHLN/table/5`, 0)
```

You can access the information using the IsDirected, IsWeighted, Vertices, Edges, Neighbors, WeightMatrix (for weighted graphs) commands.

```
> IsDirected(G);
```
false

```
> Vertices(G);
```
$$[a, b, c, d]$$

```
> Neighbors(G);
```
$$[[b, d], [a, c], [b, d], [a, c]]$$

```
> L := op(4,G);
```
$$L := \left[\{2, 4\} \ \{1, 3\} \ \{2, 4\} \ \{1, 3\} \right]$$

You can import a graph from a file in several formats e.g. dimacs, dimacs_relaxed, combinatorica, edges, and dot.
And export it in those formats.

```
> ExportGraph(G,"Ggraph",dimacs);
```

File "Ggraph" created in dimacs format

This created the following text file.

```
c Generated by the Maple GraphTheory package
p edge 4 8
e 1 2
e 2 1
e 1 4
e 4 1
e 2 3
e 3 2
e 3 4
e 4 3
```

Reading that file back into Maple we can recover the graph.

```
> H := ImportGraph("Ggraph",dimacs);
```

$H :=$ *Graph 6: an undirected unweighted graph with 4 vertices and 4 edge(s)*

And relabel the vertices 1,2,3,4 with labels *a,b,c,d*.

```
> H := RelabelVertices(H,[a,b,c,d]);
```

$H :=$ *Graph 7: an undirected unweighted graph with 4 vertices and 4 edge(s)*

```
> Neighbors(H);
```

$$[[b, d], [a, c], [b, d], [a, c]]$$

Graph drawing and special graphs

The graph drawing commands are

DrawGraph - main command
DrawNetwork - for drawing a network (must identify the source and sink)
DrawPlanar - for drawing a planar graph

The best drawing is usually obtained using the style=spring option. Here is Petersen graph and two drawings of the Petersen graph obtained using the style=spring option. The Petersen graph is one of the graphs in the **SpecialGraphs** subpackage.

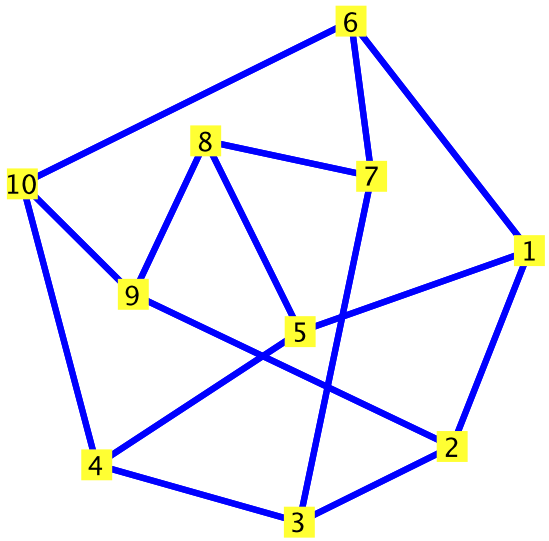
```
> with(SpecialGraphs);
```

[*AntiPrismGraph*, *CageGraph*, *ClebschGraph*, *CompleteBinaryTree*, *CompleteKaryTree*, *CoxeterGraph*, *DesarguesGraph*, *DodecahedronGraph*, *DoubleStarSnark*, *DyckGraph*, *FlowerSnark*, *FosterGraph*, *GeneralizedBlanusaSnark*, *GeneralizedHexagonGraph*, *GeneralizedPetersenGraph*, *GoldbergSnark*, *GridGraph*, *GrinbergGraph*, *GrotzschGraph*, *HeawoodGraph*, *HerschelGraph*, *HoffmanSingletonGraph*, *HypercubeGraph*, *IcosahedronGraph*, *KneserGraph*, *LCFGraph*, *LeviGraph*, *McGeeGraph*, *MobiusKantorGraph*, *OctahedronGraph*, *OddGraph*, *PappusGraph*, *PayleyGraph*, *PetersenGraph*, *PrismGraph*, *RobertsonGraph*, *ShrikhandeGraph*, *SoccerBallGraph*, *StarGraph*, *SzekeresSnark*, *TetrahedronGraph*, *ThetaGraph*, *TorusGridGraph*, *Tutte8CageGraph*, *WebGraph*, *WheelGraph*]

```
> P := PetersenGraph();
```

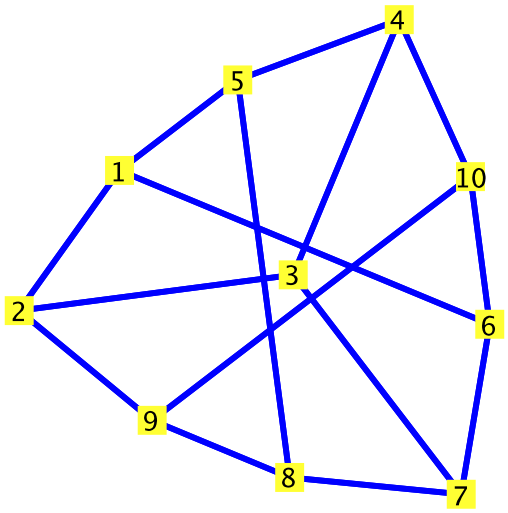
$P :=$ *Graph 8: an undirected unweighted graph with 10 vertices and 15 edge(s)*

```
> DrawGraph(P,style=spring);
```



You may have to execute these several times to get these drawings.

```
> DrawGraph(P,style=spring,redraw);
```

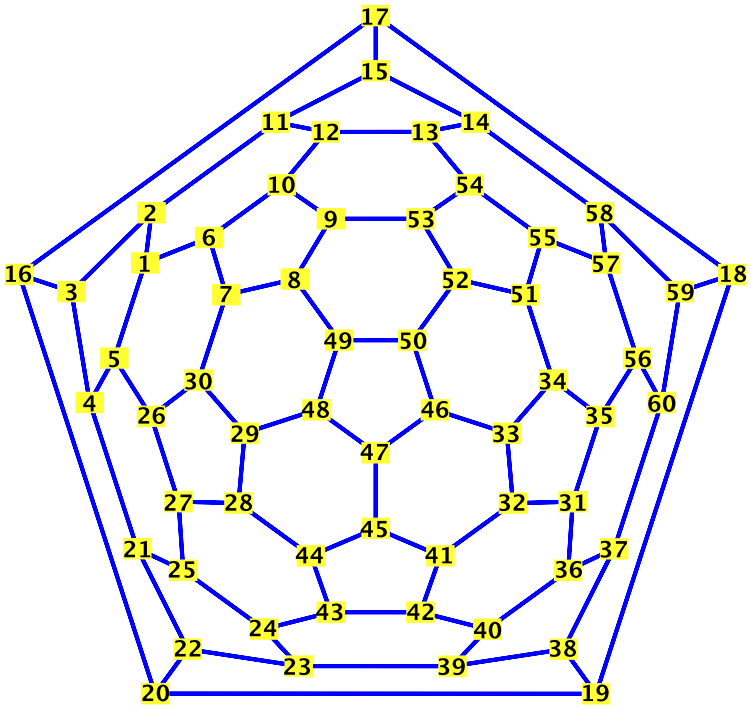


These drawings are generated by simulating a physical system where the vertices are modeled as electrons that repel each other and the edges are modeled as springs that attract each other. Initially the vertices are placed randomly in the unit square and the system (a system of differential equations) is solved to determine the final placement of the vertices. Here is the soccer ball graph (a bucky ball). In the first (and default) drawing, the vertex positions are predefined.

```
> S := SoccerBallGraph();
```

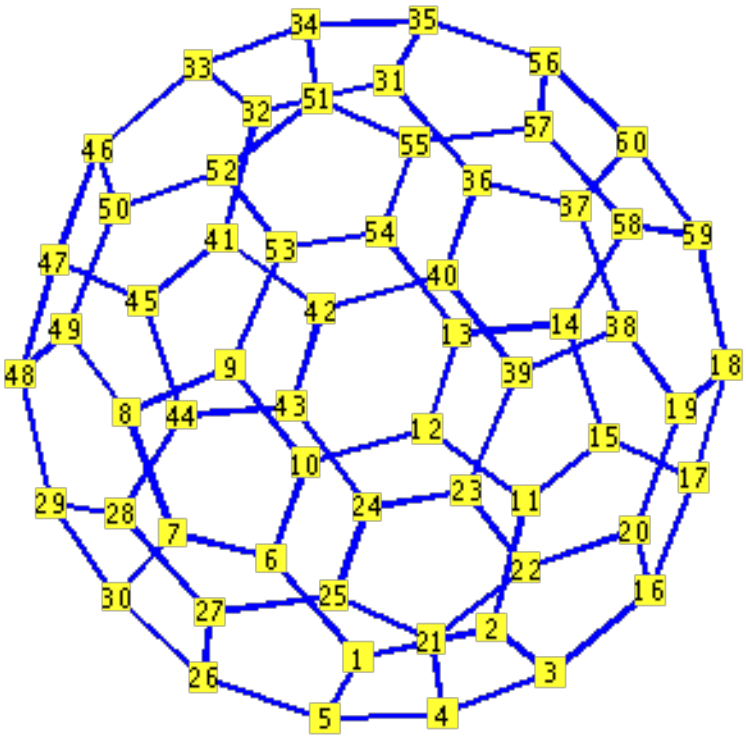
S := Graph 9: an undirected unweighted graph with 60 vertices and 90 edge(s)

```
> DrawGraph(S);
```



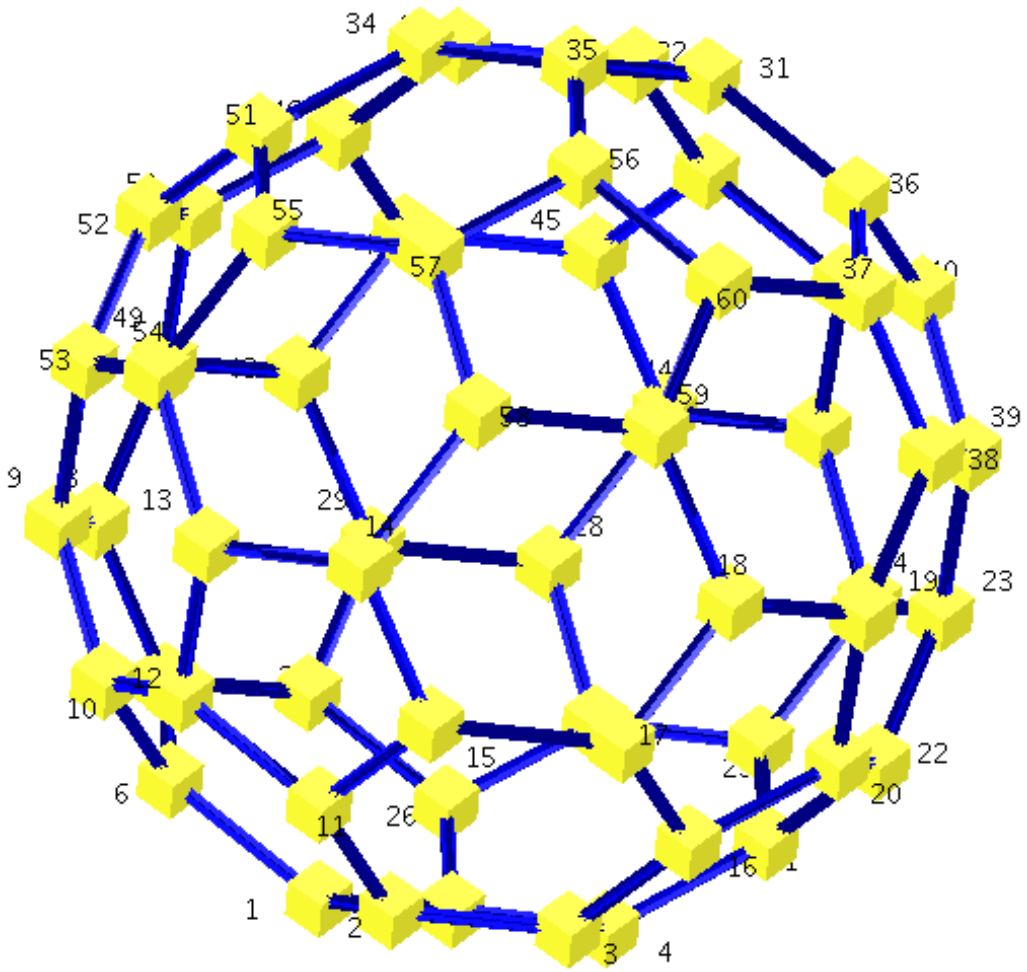
Using the style=spring option we generate this drawing.

```
> DrawGraph(S,style=spring);
```



We also support a 3D option which allows us to place the vertices in 3D and the method automatically finds a 3D representation of a soccer ball. In this version, the graph can be rotated with the mouse.

> DrawGraph(S,style=spring,dimension=3);

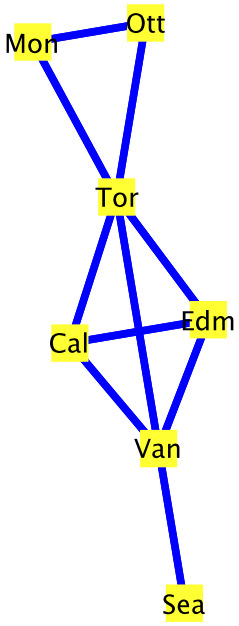


As noted, the graphs in the SpecialGraphs package have pre-defined vertex positions for drawing. You can also specify how a graph is to be drawn by specifying directly the vertex positions. Here is a Canadian example for the (mostly Canadian) cities Vancouver, Calgary, Edmonton, Toronto, Ottawa and Montreal, and Seattle. (I included Seattle because I'm a Seattle Mariners fan.)

```
> G := Graph([Van,Cal,Edm,Tor,Ott,Mon,Sea],
  {{Van,Cal},{Van,Edm},{Cal,Edm},{Cal,Tor},{Edm,Tor},
   {Tor,Ott},{Van,Tor},{Ott,Mon},{Tor,Mon},{Van,Sea}});
```

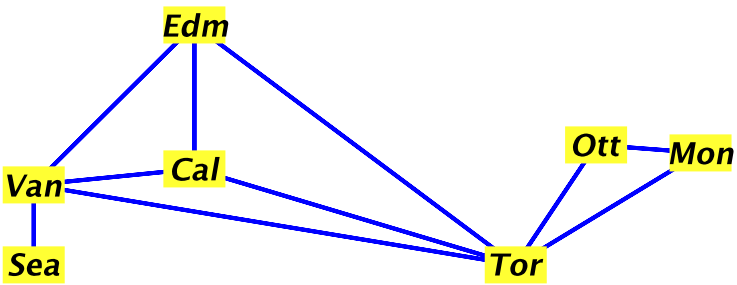
G := Graph 10: an undirected unweighted graph with 7 vertices and 10 edge(s)

```
> DrawGraph(G,style=spring);
```



Let's see if we can fix this. (The resulting figure is more faithful to the actual geographic locations.)

```
> SetVertexPositions(G,[[0,1],[2,1.2],[2,3],[6,0],[7,1.5],[8.3,1.4],
  [0,0]]);
> DrawGraph(G);
```



Graph isomorphism and graph generation

```
> with(GraphTheory):
> NonIsomorphicGraphs(6,5,restrictto=connected);
```

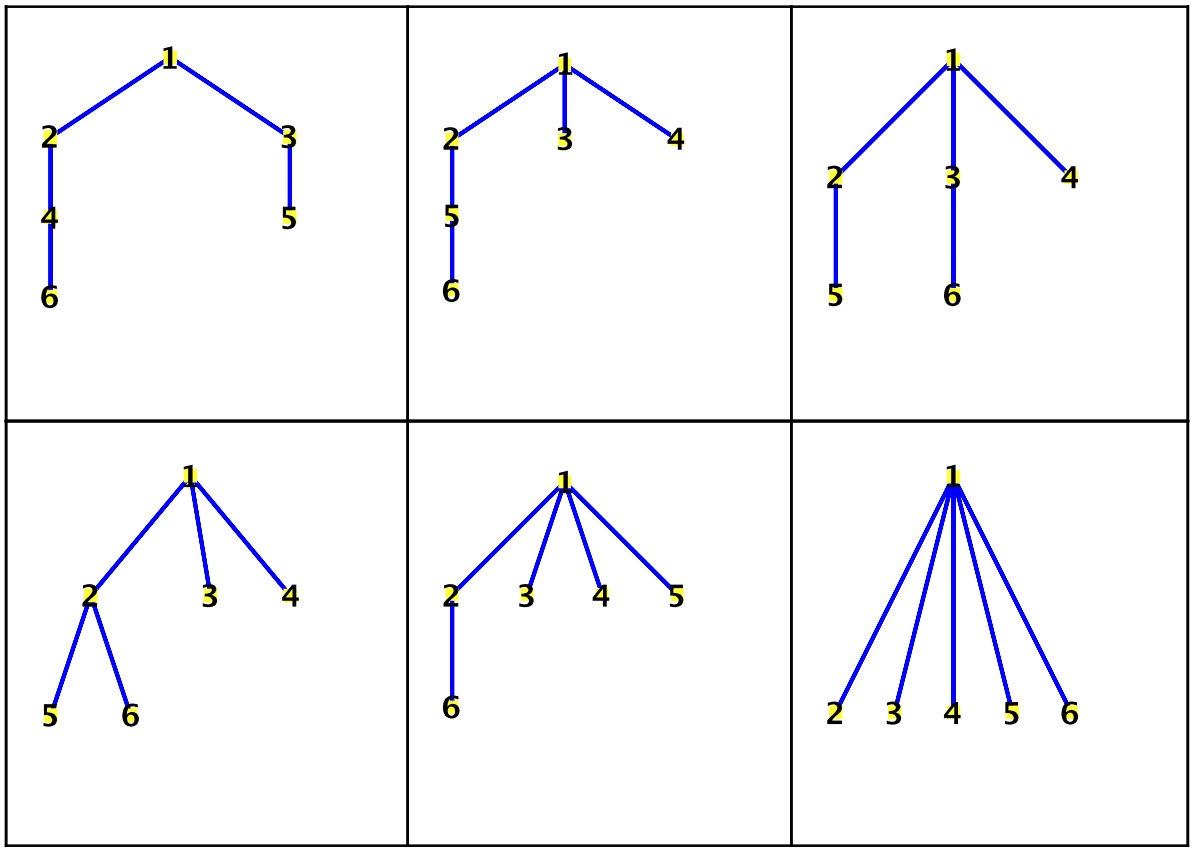
6

That means there are 6 trees with 6 vertices. We obtained *trees* by specifying $n = 6$ vertices, $m = 5$ edges and forcing the graphs to be connected. We can generate graphs for each of them and draw them side by side as follows.

```
> T := NonIsomorphicGraphs(6,5,restrictto=connected,output=graphs,
  outputform=graph);
```

T := *Graph 11: an undirected unweighted graph with 6 vertices and 5 edge(s),*
Graph 12: an undirected unweighted graph with 6 vertices and 5 edge(s),
Graph 13: an undirected unweighted graph with 6 vertices and 5 edge(s),
Graph 14: an undirected unweighted graph with 6 vertices and 5 edge(s),
Graph 15: an undirected unweighted graph with 6 vertices and 5 edge(s),
Graph 16: an undirected unweighted graph with 6 vertices and 5 edge(s)

```
> DrawGraph([T]);
```



We can restrict to regular graphs (graphs where each vertex has the same degree) as follows.

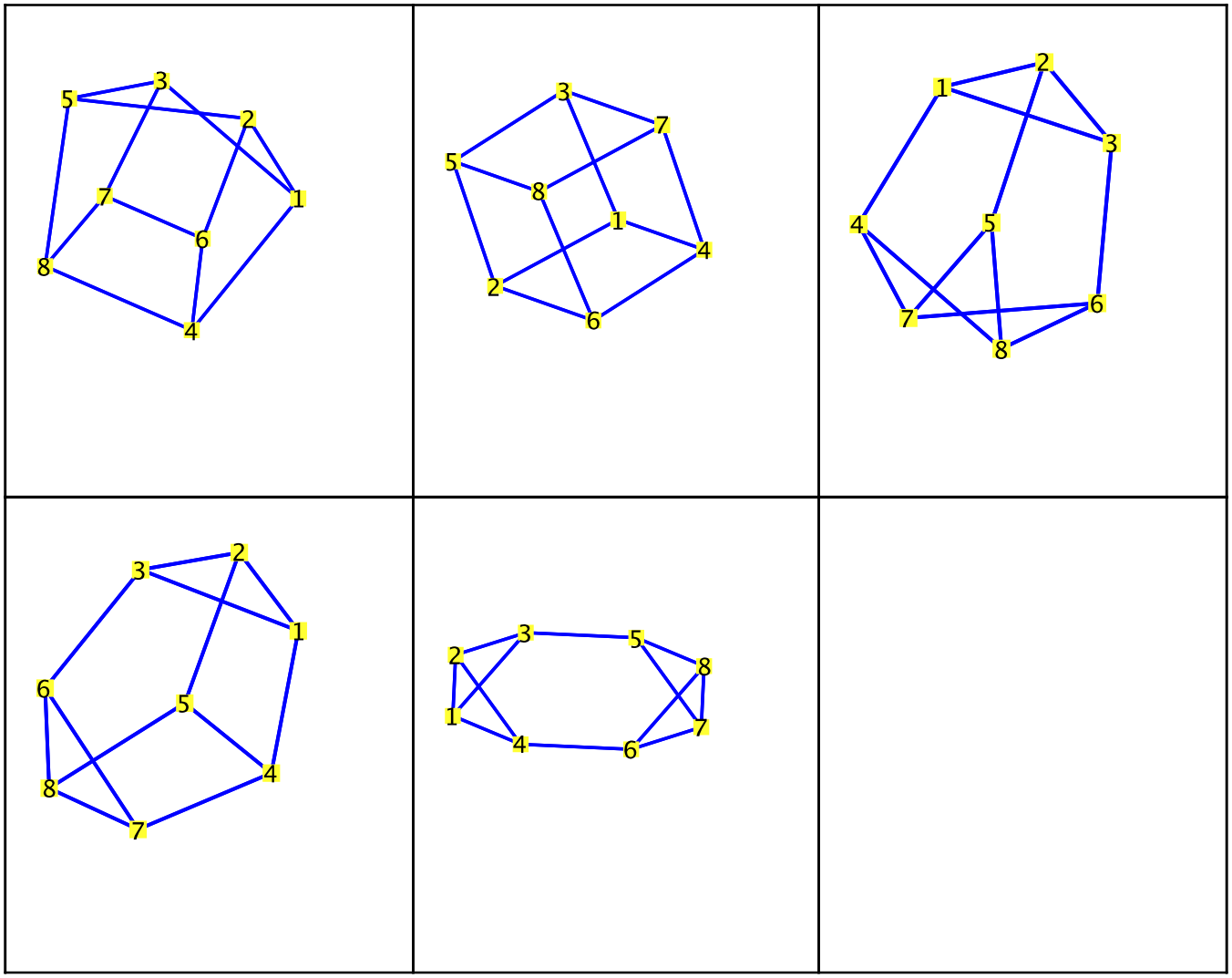
```
> NonIsomorphicGraphs(8,12,restrictto=regular);
```

6

```
> R := NonIsomorphicGraphs(8,12,restrictto=[connected,regular],output=
  graphs,outputform=graph);
```

R := *Graph 17: an undirected unweighted graph with 8 vertices and 12 edge(s),*
Graph 18: an undirected unweighted graph with 8 vertices and 12 edge(s),
Graph 19: an undirected unweighted graph with 8 vertices and 12 edge(s),
Graph 20: an undirected unweighted graph with 8 vertices and 12 edge(s),
Graph 21: an undirected unweighted graph with 8 vertices and 12 edge(s)

```
> DrawGraph([R],style=spring,width=3);
```



We can verify independently that these 5 graphs are not pairwise isomorphic as follows

```
> for g in [R] do
    for h in [R] do
      if g <> h then print(IsIsomorphic(g,h)); fi;
    od;
  od;
```

false
false
false
false
false

false

false

false

false

false

false

false

false

false

false

false

false

false

false

false

The algorithm is implemented in C. It can compute to quite large graphs. Here are the number of connected non-isomorphic graphs on $n = 10$ vertices with edges $m = 0, 1, 2, \ldots, 45$. This took less than one minute to compute on my desktop.

```
> NonIsomorphicGraphs(10,restrictto=connected,output=countbyedges);
```

$$0 = 0,\ 1 = 0,\ 2 = 0,\ 3 = 0,\ 4 = 0,\ 5 = 0,\ 6 = 0,\ 7 = 0,\ 8 = 0,\ 9 = 106,\ 10 = 657,\ 11 = 2678,\ 12 = 8548,\ 13 = 22950,\ 14 = 53863,\ 15 = 112618,\ 16 = 211866,\ 17 = 361342,\ 18 = 561106,\ 19 = 795630,\ 20 = 1032754,\ 21 = 1229228,\ 22 = 1343120,\ 23 = 1348674,\ 24 = 1245369,\ 25 = 1057896,\ 26 = 827086,\ 27 = 595418,\ 28 = 394820,\ 29 = 241428,\ 30 = 136370,\ 31 = 71293,\ 32 = 34652,\ 33 = 15767,\ 34 = 6757,\ 35 = 2768,\ 36 = 1102,\ 37 = 428,\ 38 = 165,\ 39 = 66,\ 40 = 26,\ 41 = 11,\ 42 = 5,\ 43 = 2,\ 44 = 1,\ 45 = 1$$

Random graphs

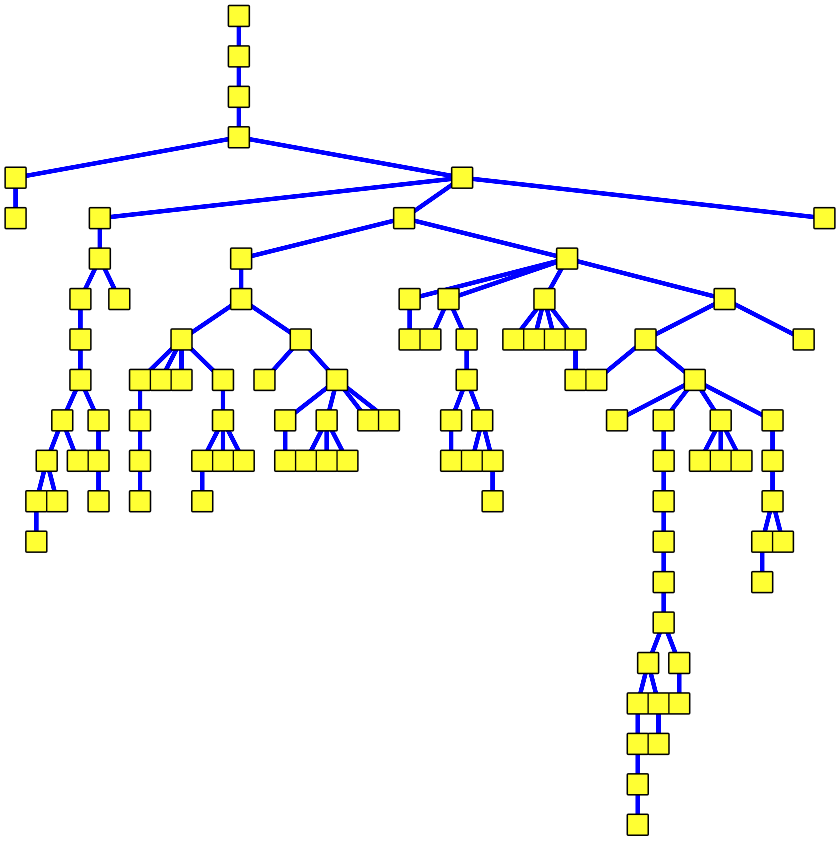
The GraphTheory package contains a subpackage of routines for generating random graphs.

```
> with(RandomGraphs);
```

[*AssignEdgeWeights*, *RandomBipartiteGraph*, *RandomDigraph*, *RandomGraph*, *RandomNetwork*, *RandomRegularGraph*, *RandomTournament*, *RandomTree*]

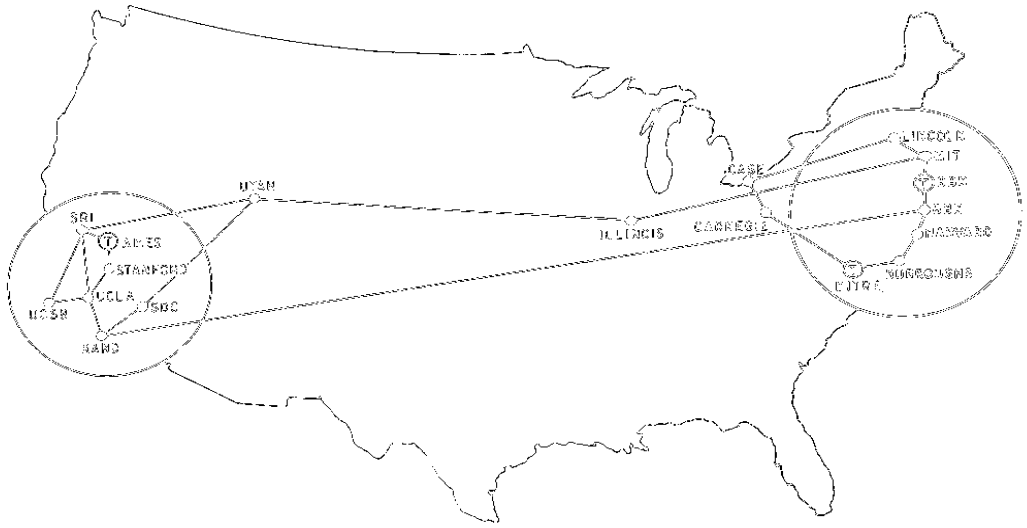
The names of the commands are self explanatory. Here is a random tree on 100 vertices. Notice that the vertex labels are suppressed by default for large graphs. I liked these drawings which look like bunches of grapes.

```
> T := RandomTree(100):
> DrawGraph(T);
```



Spanning polynomials and the reliability of computing networks

When I was a graduate student, a fellow student introduced me to *reliability polynomials* (or spanning polynomials) of networks. The idea is that if you a computer network and the edges in the network fail with probability *p*, then if you want to make the network more reliable, which new connection should you add? Consider the early internet known as the Arpanet. Each of the connections in this network is a land-line connecting computer centres.



MAP 4 September 1971

It seems to me that to improve the reliability of the Arpanet the most, we should add another connection from the western US to the eastern US. I'm going to add the edge UCSB to MITRE and try to measure how much this improves the reliability of the Arpanet. Incidentally if you google "Arpanet" you will find a sequence of images showing new connections made to the Arpanet. Let's first input the Arpanet as a graph.

```
> G := Graph(Trail(SRI,AMES,STANFORD,UCLA,UCSB,SRI),Trail(SRI,UCLA,
  RAND,SDC,UTAH,SRI),
        Trail(CASE,CARNEGIE,MITRE,BURR,HARVARD,BBN,BBNT,MIT,
  LINCOLN,CASE),
          {{RAND,BBN}},
          Trail(UTAH,ILLINOIS,MIT));
```

G := Graph 22: an undirected unweighted graph with 18 vertices and 22 edge(s)

The SpanningPolynomial(G,p) command returns a polynomial in *p* when *p* is a variable which represents the probability that each edge is operative. Thus, $p = 3/4$ means that each edge fails with probability 1/4. The spanning polynomial measures the probability that the graph G is spanned (connected). Here it is for our graph:

```
> S := SpanningPolynomial(G,p);
```

$$S := 2554\,p^{17} - 10463\,p^{18} + 17252\,p^{19} - 14318\,p^{20} + 5984\,p^{21} - 1008\,p^{22}$$

```
> eval(S,p=0.75);
```

0.186161880

Thus the graph is connected with probability 0.186.
Let's connect USCB with MITRE.

```
> AddEdge(G,{UCSB,MITRE});
```

Graph 22: an undirected unweighted graph with 18 vertices and 23 edge(s)

```
> T := SpanningPolynomial(G,p);
```

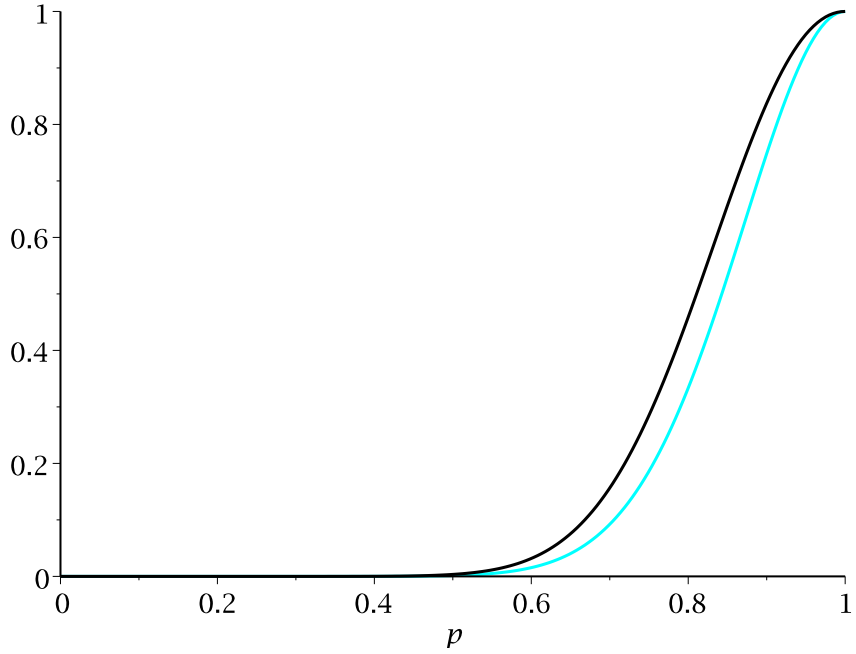
$$T := 11254\,p^{17} - 56463\,p^{18} + 118708\,p^{19} - 133858\,p^{20} + 85384\,p^{21} - 29212\,p^{22} + 4188\,p^{23}$$

```
> eval(T,p=0.75);
```

0.285019905

Not bad! Let's graph the polynomials and try to measure the improvement.

```
> plot( [S,T], p=0..1, color=[cyan,black] );
```



Notice that the black curve (with the new connection) is above the light curve. We can measure the improvement by computing the area between the two curves. That's just this definite integral

```
> int(T-S,p=0..1.0);
```

0.02978889913

Maple computes the spanning polynomial using the Tutte edge deletion/contraction algorithm mentioned previously. The method is exponential in the size of the graph so it won't work for large graphs. This is because computing the Spanning polynomial is NP-hard. There is quite a lot of literature on this subject, most of which focuses on approximation algorithms. For Maple 17, I've improved the algorithm by using a new heuristic that I called "SHARC," which stands for SHort ARC vertex ordering. The new method is still exponential but it's just 100 times faster. Using this heuristic I can compute the Tutte polynomial of the Soccer ball graph (see above) in about 2 minutes of CPU on my desktop, which compares with the previous record of 2 weeks using 150 computers by Hargard, Pearce and Royle (see [3]) below. Details of the heuristic can be found in [4] below.

▼ Teaching with the GraphTheory package

Kruskal's algorithm and Prim's algorithm

Two of the main algorithms we teach in a first course on graph theory are Prim's algorithm and Kruskal's algorithm for finding a minimal spanning tree in a weighted graph (with positive edge weights). Here is a simple example.

```
> with(GraphTheory):
> G := Graph(6,{[{1,2},2],[{2,3},3],[{3,4},2],[{1,4},3],[{4,5},2],[{5,
  6},3],[{4,6},2],[{2,5},1],[{1,6},2]});
```

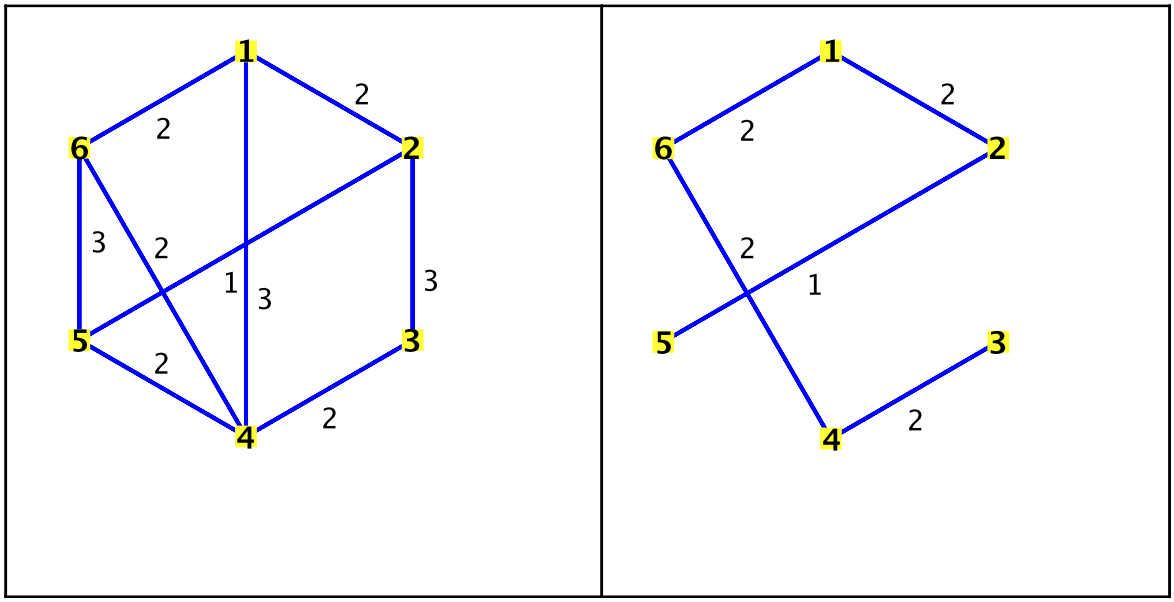
G := Graph 23: an undirected weighted graph with 6 vertices and 9 edge(s)

```
> T := MinimalSpanningTree(G);
```

T := Graph 24: an undirected weighted graph with 6 vertices and 5 edge(s)

A minimal spanning tree for the graph is a sub-tree with minimal total edge weight. In general it is not unique. The one chosen by Maple is shown on the right in the figure below. It has total edge weight 9.

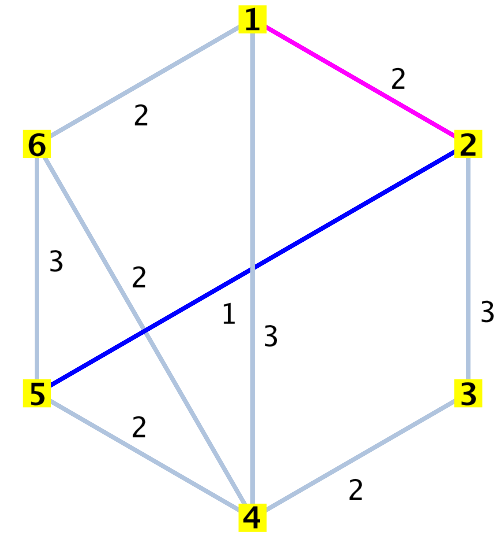
```
> DrawGraph([G,T],style=circle);
```



An animation of Kruskal's algorithm and an animation for Prim's algorithm are available using the following commands. I've shown the animation after the first step is executed. For Prim's algorithm, the edges being considered (the "fringe") are in magenta and the edge chosen is in green.

```
> KruskalsAlgorithm(G,animate);
```

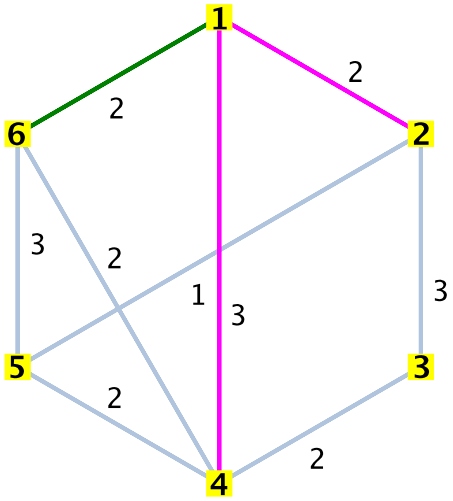
Consider the cheapest edge, {1, 2}. with weight 2



Cheapest edge
Not yet considered in spanning tree
Considered and discarded, would create a cycle

```
> PrimsAlgorithm(G,animate);
```

Cheapest fringe edge is {1, 6} with weight 2



Fringe
Cheapest edge in fringe
Not yet considered in spanning tree
Considered and discarded, would create a cycle.

References

[1] J. Farr, M. Khatarinejad Fard, S. Khodadad, and M. Monagan.
A GraphTheory Package for Maple
Proceedings of the 2005 Maple Conference, pp. 260-271, Maplesoft, 2005.

[2] M. Ebrahimi, M. Ghebleh, M. Javadi, M. Monagan, and A. Wittkopf.
A Graph Theory Package for Maple, Part II: Graph Coloring, Graph Drawing, Support Tools, and Networks.
Maple Conference 2006 Proceedings, pp. 99-112. MapleSoft, ISBN 1-897310-13-7, 2006.

[3] Gary Haggard and David Pearce.
Code for Computing Tutte Polynomials.
http://homepages.ecs.vuw.ac.nz/~djp/tutte/

[4] Michael Monagan.
A new edge selection heuristic for computing the Tutte polynomial of an undirected graph.
Proceedings of FPSAC 2012, Nagoya, Japan, July 30 - August 3, 2012.