A Graph Theory Package for Maple

Jeffrey Farr[*], Mahdad Khatirinejad[*], Sara Khodadad[*], Michael Monagan[*]

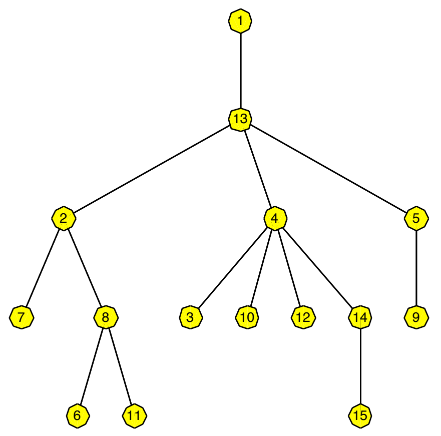
We present a new graph theory package for MAPLE. The package is presently intended for teaching and research usage, and expected to treat graphs of up to 1000 vertices in a reasonable time. The current tool in MAPLE for solving problems in graph theory is the networks package. This package is over ten years old and is designed primarily with applications of networks in mind. The data structure is too heavy and cumbersome for treating elementary graph theory problems. Therefore, one design criterion for the new GraphTheory package is a simple, yet flexible, data structure designed primarily for solving problems related to graphs rather than networks. All of the operations present in the networks package and all of the standard operations for graphs are, however, available in the GraphTheory package.

The package includes a drawing algorithm. The following example shows how one can find a spanning tree of a random graph which has 15 vertices and every edge is present with probability 0.3:

```
> G := RandomGraph(15,.3);
```

GRAPHLN(*undirected, unweighted*,[1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11, 12, 13, 14, 15], [{13} , {4, 5, 7, 8, 13} , {4, 5, 8, 9}, {2, 3, 10, 12, 13, 14} , {2, 3, 9, 12, 13, 14} , {8,12} , {2, 10} , {2, 3, 6, 11} , {3, 5, 12} , {4, 7} , {8, 15} , {4, 5, 6, 9, 14} , {1, 2, 4, 5} , {4, 5, 12, 15} , {11, 14}], *table*([]) ,0)

```
> DrawGraph(SpanningTree(G));
```



*Supported by the MITACS NCE of Canada.

We will explain the data structure. In order to construct a (di)graph, one may use the constructor commands `Graph()` or `Digraph()`.

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

$GRAPHLN(undirected, unweighted, [a, b, c], [\{2\}, \{1, 3\}, \{2\}], table([]), 0)$

The above example constructed the unweighted graph $G = (V, E)$ where $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}\}$. The data structure of a (di)graph is as follows:

$$GRAPHLN(D, W, V, A, T, EW)$$

where D and W are of type symbol, V is of type list, A is of type Array, T is of type table, and EW is of type Matrix. Two symbols, namely *directed* or *undirected*, are possible for D. Also, two symbols can be used for W, namely *weighted* or *unweighted*. The list V stores the labels of the vertices of the graph. The array A contains the set of neighbors of every vertex in an array. Notice that each element of A is of type set(posint). The table T stores some of the graph properties which are expensive to find, but are cheap to store. The matrix EW stores the edge weights of the graph when the graph is weighted. When the graph is unweighted, EW is not a matrix and is set to 0.

Some algorithmic details of the `MaximumClique`, `IsPlanar`, and `MaxFlow` commands

The problem of finding a maximum clique (or a maximum independent set) of a graph is known to be NP-complete. We have implemented a backtracking (branch and bound) algorithm which is fast (see [2]). For example on an Apple G5 machine, it takes about 2 seconds to find the maximum clique of a random graph with 1000 vertices and $250,000$ edges. The number of vertices of a maximum clique of such a graph is usually about 10 to 15. A bounding function helps to reduce the size of state space tree. The bounding function that we have used is the greedy coloring of a graph.

To test whether a graph on n vertices is planar or not, we have implemented an algorithm due to Demoucroun, et al. (see [1]), which has running time $O(n^2)$. There are some algorithms, such as Hopcroft-Tarjan algorithm, which are linear time. However, the algorithm we have used is much simpler and seems to be more efficient for graphs with less than 1000 vertices. If a graph is planar, we output the plane embedding of the graph as a set of faces. This information is obviously useful for drawing the planar graph.

The basic problem of finding a maximal flow in a network occurs not only in transportation and communication networks, but also in currency arbitrage, image enhancement, machine scheduling and many other applications. To find the maximum flow of a network on n vertices and m edges, we have implemented the so called preflow-push (push-relabel) algorithm. This algorithm runs in $O(n^2m)$ time, an improvement over the $O(nm^2)$ augmenting path algorithms, such as Edmonds-Karp, which are often used. We have also used this algorithm to find the vertex connectivity and edge connectivity of a graph.

List of commands of the GraphTheory package

AcyclicPolynomial	AddArc	AddEdge
AddVertex	AdjacencyMatrix	AllPairs
Arrivals	BiConnectedComponents	CayleyGraph
CharacteristicPolynomial	ChromaticNumber	ChromaticPolynomial
ClebschGraph	CliqueNumber	CompleteGraph
Connect	ConnectedComponents	Contract
CopyGraph	CycleBasis	CycleGraph
Deck	Degree	DegreeSequence
DeleteArc	DeleteEdge	DeleteVertex
Departures	Diameter	Digraph
Distance	DodecahedronGraph	EdgeConnectivity
Edges	FlowPolynomial	FundamentalCycle
Girth	Graph	GraphComplement
GraphDifference	GraphIntersection	GraphJoin
GraphPower	GraphRank	GraphSum
GraphUnion	GridGraph	Head
HyperCubeGraph	IcosahedronGraph	IncidenceMatrix
IncidentEdges	InDegree	IndependenceNumber
InducedSubgraph	Internal	IsAcyclic
IsClique	IsConnected	IsCutSet
IsDirected	IsEulerian	IsGraphicSequence
IsHamiltonian	IsPlanar	IsRegular
IsTree	IsWeighted	LineGraph
MakeWeighted	MaxFlow	MaximumClique
MaximumDegree	MaximumIndependentSet	MinimumDegree
MinimumSpanningTree	MycielskiGraph	Neighbors
NumberOfEdges	NumberOfTrees	NumberOfVertices
OctahedronGraph	OutDegree	PathGraph
PetersenGraph	RankPolynomial	RandomDigraph
RandomGraph	RandomTournament	SeidelSpectrum
SequenceGraph	ShortestPath	ShrikhandeGraph
SpanningPolynomial	SpanningTree	Spectrum
StandardGraph	Subdivide	Subgraph
Switch	Tail	TetrahedronGraph
TopologicSort	TravelingSalesman	TuttePolynomial
UnderlyingGraph	VertexConnectivity	Vertices

References

[1] Alan Gibbons, *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[2] D. L. Kreher, D. R. Stinson, *Combinatorial algorithms: generation, enumeration, and search.* CRC Press, 1999.