Undirected Graphs and Networks

Networks and graphs

A *network* is a collection of objects connected to each other in some specific way.

A *graph* is a finite set of dots called *vertices* (or *nodes*) connected by links called *edges* (or *connections*).

For our purposes, a *graph* and a *network* mean the same thing.

The *degree* (or *valence*) of a vertex is the number of edges attached to the vertex. A vertex of degree zero is known as an *isolated* vertex.

The sum of the degrees of all the vertices of a graph is twice the number of edges.

The number of vertices of odd degree must be even.

An edge which connects a vertex to itself is called a *loop* and contributes two towards the degree.

If two or more edges connect the same pair of vertices they are called *multiple edges* (or *parallel edges*) and all count towards the degree.

More formally: a *simple graph* is a set of vertices V and set E of unordered pairs of distinct elements of V called edges.

A network that contains multiple edges or loops is sometimes called a *multigraph*. A *simple graph* is a graph with no multiple edges or loops. The formal mathematical definition of a *simple graph* permits neither multiple edges or loops.

A graph with one vertex and no edges is called a *null graph* or a *trivial graph*.

A *complete graph* is a graph which has an edge joining every pair of vertices.

A *weighted graph* is one in which each edge is assigned a non-negative number called the *length* or *weight*. In applications, this number can represent different things such as distance, cost in dollars, time taken, etc.

A *planar graph* is a graph that can be drawn on a plane so that no edges cross.

Paths and Circuits

A *path* is a sequence of consecutive edges in a graph. We say the path *connects* the initial vertex to the terminal vertex.

A *connected graph* is one in which there is a path connecting every pair of vertices.

A *subgraph* is a part of a graph, including some or all of its vertices and edges. A *connected subgraph* is one where all the vertices of the subgraph are connected.

A *circuit* is any path in a graph that begins and ends at the same vertex.

An *Euler path* in a graph is a path that passes along each edge exactly once. (Euler is pronounced as in "boiler"). An *Euler circuit* is a circuit which passes along each edge exactly once.

A graph has an Euler *circuit* if and only if it is connected and every vertex is of even degree. A graph can have an Euler *path* only if either every vertex is of even degree or if exactly two vertices are of degree one. In the latter case, the Euler path must start and end on the two vertices of degree one. A connected graph that has a Euler path but no Euler circuit is called *semi-Eulerian*.

Note that a Euler path or circuit may have repeated vertices.

In most cases, Euler circuits can be found easily by just using a couple of commonsense guidelines: (a) always leave one edge available to get back to the starting vertex as the last step, and (b) don't use an edge to go to a vertex unless there is another edge available to leave that vertex (except for the last step). There is an algorithm guaranteed to find an Euler circuit that involves finding and inserting connected circuits until all edges have been used. *Fleury's algorithm* will also find an Euler circuit in a connected graph.

Practical examples involving Euler paths and circuits include the route inspection problem and the problem of the Seven Bridges of Königsberg. What route will take the garbage truck down every street in the town without going along the same street twice? Is it possible to walk a route that crosses each bridge in Königsberg (now Kaliningrad) exactly once and returns to the starting point?

A *Hamiltonian path* is a path in a graph which passes through every vertex exactly once. A *Hamiltonian circuit* is a circuit which passes through every vertex exactly once. For a Hamiltonian path to exist in a graph, there can be up to two vertices of degree one (dead ends), but there is no general rule that allows us to determine whether a graph has a Hamiltonian circuit.

Note that a Hamiltonian path or circuit need not use all of the edges of a graph.

The *Traveling Salesman Problem* is the problem of finding a minimal cost Hamiltonian circuit on a weighted graph. What route should the salesman take to visit every town just once at minimum cost?

There is no efficient algorithm to solve the traveling salesman problem apart from the brute-force method of listing all possible Hamiltonian circuits and calculating the total cost for each. The *Nearest neighbour algorithm* will usually give an approximate solution, but not necessarily the optimal one. Pick a starting vertex, and form a circuit which starts and ends at that vertex by always traveling along the edge coming out of your vertex that has the lowest weight, being careful to never visit a vertex twice until you have been to all of them once.

Trees

A *cycle* is a subgraph whose path contains at least three vertices and whose start and end vertices are the same.

A *tree* is a connected graph that has no loops, multiple edges, or cycles.

The shortest path between two vertices in a graph is always a tree. *Dijkstra's algorithm* will find the minimum distance path (see below).

A *spanning tree* on a graph is a subgraph that contains all of the vertices of the graph and is a tree. A *minimum-cost spanning tree* on a weighted graph is a spanning tree that has the lowest total cost - that is, a spanning tree such that the total of all of the numbers on the edges used is as small as possible. The minimum-cost (or *maximum-cost*) spanning tree can be found simply using *Prim's Algorithm*: start by selecting the edge with the least weight in the graph and add that to the empty tree T, then select the remaining edge of least weight that is connected to one of the vertices of T and will not create a cycle and add that to T. Repeat until all vertices are included in T.

One application of minimum-cost spanning trees would be designing an efficient telephone network. The vertices represent places you need to connect to the network and the edges represent phone lines you *could* build. The weights of the edges represent the costs of building the corresponding line. A minimum-cost spanning tree is the design that allows you to connect everyone to the network for the least cost.

Dijkstra's algorithm involves finding a spanning tree that gives the minimum weight from the starting vertex to each vertex in the graph. Start with an empty tree T. Add the edge connected to the starting vertex that has the least weight. Consider each edge connected to T that could be added to it without creating a cycle and calculate the total weight from the starting point to the new vertex that would be connected. Choose the edge that gives the lowest weight and add that to T. Repeat until all vertices have been added to T or until the required end vertex has been added.

References

Nolan, J, et al. *Maths Quest 12: Further Mathematics VCE Mathematics Units 3 and 4*, John Wiley & Sons Australia, 2000.

Grossman, Peter. *Discrete Mathematics for Computing*, 2nd ed., Palgrave Macmillan, 2002.

Lipschultz, Seymour and Lipson M, *Schaum's Easy outline: Discrete Mathematics*, McGraw-Hill, 2003.

University of Nebraska Lincoln. *Math 203 Review for Exam 1*, <http://www.math.unl.edu/~jwalker/m203inst/ch123rev.pdf>.

Bowen, Dr. Larry. *MATH 103 Introduction to Contemporary Mathematics*, <http://www.ctl.ua.edu/math103/>

Locke, Stephen C., *Graph Theory*, <http://www.math.fau.edu/locke/graphthe.htm>.

Wikpedia, *Graph theory*, <http://www.wikipedia.org/wiki/Graph_theory>.

