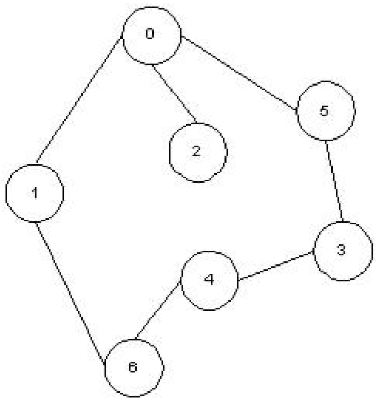
A Simple Application of Graph Theory

A graph is simply a set of vertices (or endpoints) and a set of edges that connect some or all of the vertices.

Figure 1 illustrates a basic graph structure.

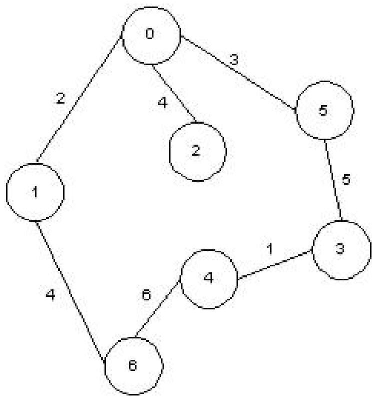


The vertices in Figure 1 are the numbered circles, and the edges are the lines joining the circles. A vertex can represent any entity that you want to model; for example, a vertex might represent a geographical location, and the edges might represent routes between those locations. Vertices that are joined are said to have an adjacency with each other.

So how does all this relate to the real world?

Imagine that the graph in Figure 1 represents geographical locations or nodes in a network. Further, let's assign each edge a cost—the higher the cost the more expensive the route.

I've added some costs (or weights) in Figure 2 (A graph with weighted edges).



Now, to get the best route from one node to another, you add up the weights of the edges traversed. The journey with the lowest overall weight is the cheapest; to get from 0 to 4 via vertices 5 and 3 incurs a cost of 3 + 5 + 1 or 9.

To get from 0 to 4 via vertices 1 and 6 incurs a cost of 2 + 4 + 6, or 12. So we can say that to get from 0 to 4, the cheapest route is 0-5-3-4.

Clearly, this is a very small example, and you can imagine the problems faced using graphs to represent very large data sets: traversal becomes expensive; storage requirements shoot up, and so on. But, you get the idea.

Once you start to understand these simple concepts you're beginning to get the basics of graphs. Bear in mind that there is an enormous amount of theory behind this area, but it is possible to gain an overview.

Moving up the value chain is all about being able to tackle complex problems and subjects (see references [3] and [4] for more information).

Let's now look at some simple Java to help us to program some concrete examples.

Listing 1 illustrates an excerpt from a Java class called Graph.

Listing 1 A large part of the Graph class

```
private int VertexCount, EdgeCount;
  private boolean digraph;
  private boolean adjacencies[][];
  public GraphDetails graphDetails;

  Graph(int numVertices, boolean flag)
  {
    VertexCount = numVertices;
    EdgeCount = 0;
    digraph = flag;
    adjacencies = new boolean[VertexCount][VertexCount];

    graphDetails = new GraphDetails();
  }

  int numVertices() { return VertexCount; }
  int numEdges() { return EdgeCount; }
  boolean directed() { return digraph; }
```

- *VertexCount* represents the number of vertices in the graph
- *EdgeCount* represents the number of edges in the graph
- *digraph* indicates whether the vertices are directional in nature
- *adjacencies* represents the adjacencies information in the graph
- *GraphDetails* just loops through the graph and prints out the vertex and edge details

Listing 1 is the constructor for a Java class called Graph. To create the graph shown in Figures 1 and 2, you instantiate an object of the Graph class with a line something like this:

```
Graph myGraph = new Graph(7, false);
```

The preceding code should help in getting in on the ground floor of graph programming! Rather than having to study a lot of theory, the example code and the following explanations should provide an onramp to this overtly complex area of programming. The object construction just creates the graph and tells the object that it will have seven vertices. The edges must be added after this by using calls such as the following:

```
myGraph.insertEdge(new Edge(0, 1));
myGraph.insertEdge(new Edge(0, 2));
```

The above two lines of code create the edges from vertex 0 to 1 and vertex 0 to 2, respectively.

It's not hard to see that the other edges in Figures 1 and 2 are created simply by additional calls to the method:

```
myGraph.insertEdge().
```

Let's now take a look at how the graph looks once it's been built.

The complete graph in Figures 1 and 2 can be created and displayed using the code in Listing 2.

Listing 2 Building the graph

```
Graph myGraph = new Graph(7, false);
myGraph.insertEdge(new Edge(0, 1));
myGraph.insertEdge(new Edge(0, 2));
myGraph.insertEdge(new Edge(0, 5));
myGraph.insertEdge(new Edge(5, 3));
myGraph.insertEdge(new Edge(3, 4));
myGraph.insertEdge(new Edge(4, 6));
myGraph.insertEdge(new Edge(6, 1));

myGraph.graphDetails.show(myGraph);
```

Once the Listing 2 code runs, you should see the output shown in Listing 3 appear.

Listing 3 Displaying the graph contents

```
Graph details
0: 1 2 5
1: 0 6
2: 0
3: 4 5
4: 3 6
5: 0 3
6: 1 4
```

Listing 3 can be read as follows: the left number corresponds to a given vertex. The numbers after the colon indicate the vertices adjacent to the left hand vertex.

So, vertex 0 is adjacent to vertices 1, 2, and 5, respectively; as you can verify from Figures 1 and 2.

… The more will continue later if you are interested…