1 What is Graph Theory?

Graph theory is the study of mathematical objects known as *graphs* (and these are not your typical bar and pie charts!). It is a more theoretical topic than others we've discussed so far, and it is a type of discrete math I've studied a lot (so you better like it!). We've already seen one type of graph in 211, and that is the *tree*.

Why Should I Care?

Graph theory, despite being very theoretical, has many applications in a wide variety of [computer science-related] fields, such as artificial intelligence, compilers, numerical analysis, circuit design, scheduling, etc. Besides, I already told you that I like it a lot, and you wouldn't want to hurt my feelings... right?

2 Definitions & the Nitty Gritty

Graph A *graph* G is a pair (V, E) where V is a finite set of objects called *vertices* (singular: *vertex*) and E is a set of unordered pairs of distinct vertices called *edges*, i.e. edges connect pairs of vertices.

Adjacent If $e = \{v_i, v_j\}$ is an edge in our graph then we say that v_i and v_j are *adjacent* vertices.

Indcident If $e = \{v_i, v_j\}$ is an edge in our graph then we say that the edge e is *incident* with vertices v_i and v_j. We can also say an edge *joins* v_i and v_j.

Example 1: Suppose $V(G) = \{v_1, v_2, v_3, v_4\}$, and $E(G) = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{v_2, v_4\}\}$. Then our graph would look like:

We would say that v_1 is adjacent to v_2 and v_3, v_2 is adjacent to v_1, v_3, and v_4, v_3 is adjacent to v_1 and v_2, and v_4 is adjacent to v_2. Furthermore, edge $\{v_1, v_2\}$ is incident to v_1 and v_2, etc.

Example 2: The Word Graph

The word graph W_n is the graph having $V(W_n)$ being the set of all English words having exactly n letters, and edges defined by: two words are adjacent if one can be obtained from the other by replacing exactly one letter by another (in the same position). I.e. 'seat' is adjacent to 'sent' in W_4

Example 3: The Street Map Graph

Given a street map of a city, like Ithaca, one can define a street map graph as follows: there is a vertex for each intersection, and an edge for each part of a street joining 2 intersections and traversing no other intersection. Such graphs are useful in solving certain kinds of routing and scheduling problems, such as garbage pickups, delivery of newspapers, bus scheduling, etc.

Example 4: The World Map Graph

Another way to get a graph from a map: suppose you start with a map of countries of a continent, or states in the USA. There is a vertex for each country and 2 countries are adjacent (i.e. there's an edge) if they share a boundary. One of the most famous problems in graph theory arose from the question of how many colours are needed to colour maps so that adjacent countries (states) are not assigned the same colour. In fact, it has been proven you can colour any map with only 4 colours, without any adjacent countries being the same colour.

Degree The number of edges incident with a vertex v is called the *degree* of v, denoted $\deg(v)$.

Path A *path* in a graph is a sequence of distinct vertices $w_1, w_2, \ldots, w_n$ such that (w_i, w_{i+1}) is an edge, for $i = 1, \ldots, n-1$. The *length* of such a path is the number of edges on the path, namely $n-1$. Formally, we allow paths of length 0, namely a path on one vertex using *no* edges.

Cycle A *cycle* in a graph is a path of length at least 1 such that $w_1 = w_n$. An *acyclic* graph has no cycles.

Connected A graph G is *connected* if and only if there is a path between any two vertices in G.

Directed Graph A *directed graph*, or *digraph* is a pair (V, E) where V is as in the definition of graph, and E is a set of *ordered* pairs of distinct vertices called *edges*. In other words, the edges are directed (so edge (v, w) is different from edge (w, v)).

Example 5: An Airport System

Suppose each airport is a vertex, and if there is a nonstop flight between correspond airports, then the two vertices are connected by an edge. The edge could have a "weight", representing time, distance, or the cost of the flight. Generally, an edge $(v, w) \Rightarrow (w, v)$, but this isn't always the case. Maybe we'd like to find the best flight between 2 airports; "best" being fastest or cheapest, ...

Network A *network* is a graph together with zero or more functions from the nodes to the real numbers and zero or more functions from the edges to the real numbers. Usually we only consider the functions from edges. In other words, each edge has a numeric 'cost' associated with it.

Example 6: Routing Email

Suppose the vertices of our graph represent computers and the edges represent links between pairs of computers. The edge costs represent communication costs (phone bills per megabyte), delay costs (seconds per megabyte), or a combination of these and other factors.

3 The Unweighted Shortest Path Problem

Here we would like to consider the problem of finding the shortest path between two specified vertices.

3.1 The Problem Statement

Find the shortest path (measured by number of edges) from a designated vertex s (called the source) to every vertex. If there is no path to a given vertex in the directed graph, then that 'path' has length infinity. (Question to *you*: When is there no path?)

3.2 The Solution

The rough idea to solve the unweighted shortest path problem is as follows: First we find all vertices of shortest path length 0 from s, then all those of length 1 from these (path length 1), then all those of length 1 from those (path length 2), and so on. For example, consider the following graph:

length from s	0	1	2	3	4
which vertices	s	1, 2	3, 4, 5	6	7

The algorithm to solve the unweighted shortest path problem is:

```
Keep a set R of vertices already visited in our algorithm.  We'll add to R.
R is initialized to contain only the vertex s, i.e. R={s}
Every vertex v keeps track of its distance from s (v.dist) and the vertex p
that precedes it on its shortest path (v.pred):
  - Set s's distance to be 0. (s.dist = 0)
  - Set s's preceding vertex to be no one. (s.pred = null)

while R not equal to all of V
    for each vertex r in R do
        // we have already visited r, and have calculated its distance and preceding vertex
        for each vertex v adjacent to r
            if v is not already in R
                // then we haven't calculated v's distance from s yet
                set v's distance:  v.dist = 1 + r.dist
                set v's preceding vertex: v.pred = r
                add v to R, since we've now visited it
            endif
        endfor
    endfor
endwhile
```

Excercise: What is the running time of the above algorithm?

The details of the implementation of the unweighted shortest path solution are at the end of this package.

4 The Weighted Shortest Path Problem

Now we would like to consider the variation of the unweighted shortest path problem where the edges of the network have weights, which for now we will assume are non-negative.

4.1 The Problem Statement

Find the shortest path (measured by total cost) from a designated vertex s to every vertex. All edge costs are assumed non-negative.

4.2 The Solution: Dijkstra's Algorithm

Dijkstra's algorithm efficiently solves the shortest-path problem, assuming that the edge costs are non-negative. For each node v, $v.dist$ denotes the cost c of a shortest path of from s to v in G. The main idea of the algorithm is similar to the unweighted solution: Suppose that for some node set $R \subseteq V$ we know the value $r.dist$ for each $r \in R$. Again we'll have $s \in R$ and $s.dist = 0$. For each node $v \in V \backslash S$ (the unknown, unvisited vertices), we compute a "temporary label" $l(v)$, where

$$l(v) = \min \{r.dist + c(rv) \ : \ r \in R, \ rv \in E\}$$

(If no edge exists joining a node of R to v, then we define $l(v) = \infty$.) So $l(v)$ is the least-cost path from s to v using only the vertices in R. We then select the $v \in V \backslash R$ that has the smallest such $l(v)$, and add it to the known set R and update the labels. We then continue trying to add vertices to R, always adding the lowest-cost label vertex next.

The algorithm to solve the weighted shortest path problem is:

```
Keep a set R of vertices already visited in our algorithm.  We'll add to R.
R is initialized to contain only the vertex s, i.e. R={s}
Every vertex v keeps track of its cost from s (v.dist) and the vertex p
that precedes it on its shortest (least-cost) path:
  - Set s's distance to be 0. (s.dist = 0)
  - Set s's preceding vertex to be no one. (s.pred = null)
Initialize all labels l(v) = infinity for v not in R (because so far the paths are infinity)

while R not equal to all of V
    for each vertex r in R do
        // we have already visited r, and have calculated its distance and preceding vertex
        for each vertex v adjacent to r
            if v is not already in R AND l(v) > r.dist + c(rv)
                // then we have a less-cost label to v than before, and it goes through r
                set l(v) = d(r) + c(rv)
                set v's preceding vertex: v.prec = r
            endif
        endfor
        // add node v in V\R with the smallest l(v) to R
        find a node y in V\R with l(y) = min { l(v) : v in V\R }
        add y to R
        set y.dist = l(y)
    endfor
endwhile
```

Example 7: The Weighted Shortest Path Problem

step 0	step 1	step 2	step 3	step 4
R = {s} s.dist = 0 s.pred = null	l(a) = 4; a.pred = s l(b) = 8; b.pred = s select a a.dist = 4 add a to R	l(c) = 7; c.pred = a l(b) = 6; b.pred = a select b b.dist = 6 add b to R	l(d) = 11; d.pred = b select d d.dist = 11 add d to R	l(c) = 7; c.pred = a select c c.dist = 7 add c to R

Great, so we have a solution. *But*, Dijkstra's algorithm will fail if we allow negative costs in our network, because there will be lower cost paths appearing later in the algorithm going back to previously marked 'known' vertices. There are solutions to this problem as well, which we will not cover.

5 Representing a Graph

You need to be able to represent a graph in a computer, and to manipulate its vertices, edges, and costs in an efficient manner. There are a few common ways to do this, which we will discuss here.

5.1 Adjacency Matrix

An *adjacency matrix* $= (a_{ij})$ of a graph G has its rows and columns indexed by $V(G)$ and is defined by

$$a_{ij} = \begin{cases} c(v_i v_j) & \text{if node } i \text{ and } j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

Example 8: An Adjacency Matrix of Example 7

$$A_{ij} = \begin{pmatrix} 0 & 4 & 8 & 0 & 0 \\ 4 & 0 & 2 & 3 & 0 \\ 8 & 2 & 0 & 0 & 5 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \end{pmatrix}$$

Note: An undirected graph has a symmetric adjacency matrix.

5.2 Adjacency List

A graph that does not have a lot of edges (i.e. not near the maximum) is called *sparse* (as opposed to *dense*). For a sparse graph, our adjacency matrix will be mostly 0's (a.k.a. *sparse* in matrix terminology, what a coincidence!), so it is a waste of space to keep all of them. You can imagine that it would be more efficient to just keep the non-zero values of the adjacency matrix. We do this by using an *adjacency list*. For each vertex we keep a linked list of all adjacent vertices.

Example 9: An Adjacency List
The adjacency list for the above adjacency matrix is:

0 : 1 2
1 : 0 2 3
2 : 0 1 4
3 : 1
4 : 2

6 Implementing a Graph

6.1 Class Vertex

The implementation of a vertex could be as follows:

```
public class Vertex {
   String      label;       // label of the vertex
   VertexList  adjacency;   // vertices adjacent to this one.
   Vertex      predecessor; // previous vertex on shortest path
   int         dist;        // used for the cost in the shortest path problemos
   int         inDegree;    // number of edges coming in to node

   // Constructor
   public Vertex( ) {
      this.label     = "";
      this.adjacency = new VertexList( );
      this.dist      = Graph.INFINITY;
   }

   public Vertex( String label ) {
     this( );
     this.label = label;
   }

   public Vertex( String label, VertexList adjacency ) {
      this( label );
      this.adjacency = adjacency;
   }


   // Getter methods
   public String     getLabel( )               { return label; }
   public VertexList getAdjacency( )           { return adjacency; }
   public Vertex     getPredecessor( )         { return predecessor; }
   public int        getDistance( )            { return dist; }


   // Setter methods
   public void   setLabel( String label )       { this.label = label; }
   public void   setAdjacency( VertexList adj ) { this.adjacency = adj; }
   public void   setDistance( int dist )        { this.dist  = dist; }
   public void   setPredecessor( Vertex pred )  { this.predecessor = pred; }

   // Other methods
   public boolean equals( Vertex v ) {
      return ( label.equals( v.getLabel() ) );
   }

   public String toString( ) {
     return label;
   }
}
```

6.2 Class VertexList

```
public class VertexList {
    Vertex[] list;
    int      length;
    int      current;

    public VertexList( ) {}

    public VertexList( int numVertices ) {
      list = new Vertex[ numVertices ];
      for( int i = 0; i < numVertices; i++ ) {
         list[ i ] = new Vertex( "" + i );
      }
      length  = numVertices;
      current = numVertices;
    }

    public String toString( ) {
      String s = new String( "" );
      for( int i = 0; i < length; i++ ) {
        s = s + list[ i ] + " ";
      }
      return s;
    }

    public Vertex get( int i ) {
      return list[ i ];
    }

    public void set( int i, Vertex v ) {
      if( i < length )
        list[ i ] = v;
      if( v == null )
        current--;
    }

    public void add( Vertex v ) {
      if( current < length )
        list[ current++ ] = v;
    }

    public int getLength( ) { return length; }

    public int getCurrent( ) { return current; }

    public void clear( ) {
      for( int i = 0; i < length; i++ ) {
        list[ i ] = null;
      }
    }
}
```

6.3 Class Graph

```
public class Graph {
  public static final int INFINITY = Integer.MAX_VALUE;
  int        numNodes;
  int        numEdges;
  VertexList vertexList;      // List of vertices in graph
  int[][]    adjacencyMatrix; // Adjaceny matrix of graph

  public Graph( int numNodes ) {
    this.numNodes   = numNodes;
    vertexList      = new VertexList( numNodes );
    adjacencyMatrix = new int[ numNodes ][ numNodes ];
  }

  public Graph( String fname ) throws IOException{
       BufferedReader in = new BufferedReader( new FileReader( fname ) );
       String tmp;

       // Get the number of nodes in the graph
       tmp = in.readLine( );
       if( tmp == null )   {
          numNodes = 0;
          return;
       }
       Integer tmpInt  = new Integer( tmp );
       numNodes        = tmpInt.intValue();

       // Initialize the adjacency matrix and adjacency list
       adjacencyMatrix = new int[ numNodes ][ numNodes ];
       vertexList      = new VertexList( numNodes );

       tmp = in.readLine();
       int i = 0;
       while( tmp != null ) {
         // create the i'th vertex
         Vertex current = vertexList.get( i );
         StringTokenizer st = new StringTokenizer( tmp );
         int j = 0;
         VertexList adjList = new VertexList( numNodes - 1 );
         while( j < ( numNodes - 1 )  ) {
           if( st.hasMoreTokens() ) {
              Integer blah = new Integer( st.nextToken() );
              adjList.set( j, vertexList.get( blah.intValue() ) );
           } else
             adjList.set( j, null );
            j++;
         }
         current.setAdjacency( adjList );
         tmp = in.readLine();
         i++;
       }
  }
```

```
  public int getNumNodes( )                     { return numNodes; }
  public int getNumEdges( )                     { return numEdges; }
  public VertexList getVertexList( )            { return vertexList; }
  public int[][] getAdjacencyMatrix( )          { return adjacencyMatrix; }

  public void setNumNodes( int nodes )          { numNodes = nodes; }
  public void setNumEdges( int edges )          { numEdges = edges; }
  public void setVertexList( VertexList vlist ) { vertexList = vlist; }
  public void setAdjacencyMatrix( int[][] m )   { adjacencyMatrix = m; }

  public String toString( ) {
    return vertexList.toString( );
  }

  public void unweighted( ) {
    if( numNodes > 0 )
      unweighted( vertexList.get( 0 ) );
  }

  public void unweighted( Vertex s ) {
    LinkedList q = new LinkedList( );
    Vertex     v = new Vertex();
    Vertex     w = new Vertex();

    q.addFirst( s );
    s.setDistance( 0 );

    while( !q.isEmpty() ) {
       v = (Vertex)q.removeLast( );
       VertexList adj = v.getAdjacency( );

       // For each w adjacent to v
       for( int i = 0; i < adj.getCurrent(); i++ ) {
         w = adj.get( i );
         if( w.getDistance() == Graph.INFINITY ) {
            w.setDistance( v.getDistance() + 1 );
            w.setPredecessor( v );
            q.addLast( w );
         }
       }
    }
    printPredecessor();
  }

  public void printPredecessor( ) {
    String tmp = new String( "" );
    for( int i = 0; i < numNodes; i++ ) {
      Vertex v = vertexList.get( i );
      tmp = tmp + v.getPredecessor() + " ";
    }
    System.out.println( "Predecessor vector " + tmp );
  }
}
```