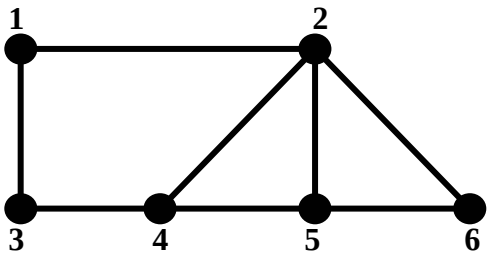
CMPS 101
Algorithms and Abstract Data Types
Winter 2006

Programming Assignment 3
Breadth First Search and Shortest Paths in Graphs

Due Monday February 13 10:00 pm

The purpose of this assignment is to implement a Graph ADT and associated algorithms in java. This project will utilize your List ADT from programming assignment 1, so spend some time going over the grader's comments from that assignment to make sure your List is working properly. Begin by reading appendices B.4 and B.5 (p. 1080-1091) and sections 22.1 and 22.2 (p. 527–539) from the text. The adjacency list representation of a graph consists of an array of lists. Each list corresponds to a vertex in the graph and gives the neighbors of that vertex. For example, the graph



has adjacency list representation

```
1: 2 3
2: 1 4 5 6
3: 1 4
4: 2 3 5
5: 2 4 6
6: 2 5
```

You will create a Graph ADT which represents a graph as an array of Lists. Each vertex will be associated with an integer label in the range 1 to *n*, where *n* is the number of vertices in the graph. Your program will use this Graph ADT to find shortest paths (i.e. paths with the fewest edges) between pairs of vertices. The client program which uses your Graph ADT will be called `FindPath`, and will take two command line arguments (here `%` denotes the unix prompt):

```
% FindPath input-file output-file
```

In this project, as in pa1, you are to write a Makefile which places your `.class` files into an executable `jar` file called `FindPath`, which makes it possible to leave out `java` on the command line. You can easily alter the Makefile from pa1 to work in this project by replacing the line `MAINCLASS = Shuffle` with `MAINCLASS = FindPath`. If you'd like to learn more about Makefiles, I again suggest you go to my CMPS 12B/12M website from Spring 2005 (http://www.soe.ucsc.edu/classes/cmps012b/Spring05/) and read Lab Assignment 1, as well as some of the Makefiles under the 'examples' link. See also Lab Assignment 2 from that class to learn how to read command line arguments and perform input-output operations to and from files in java. Also read the example `FileIO.java` on our website.

File Formats

The input file will be in two parts. The first part begins with a line consisting of a single integer *n* which gives the number of vertices in the graph. Each subsequent line will represent a single edge in the graph by a pair of distinct numbers in the range 1 to *n*, separated by a space. These numbers are the end vertices of the corresponding edge. The first part of the input file defines the graph, and will be terminated by a dummy line containing "`0 0`". After these lines are read your program will print the adjacency list representation of the graph to the output file. For instance, the lines below define the graph pictured above, and cause the above adjacency list representation to be printed.

```
6
1 2
1 3
2 4
2 5
2 6
3 4
4 5
5 6
0 0
```

The second part of the input file will consist of a number of lines, each consisting of a pair of integers in the range 1 to *n*, separated by a space. Each line specifies a pair of vertices in the graph; a starting point (or source) and a destination. The second part of the input file will also be terminated by the dummy line "`0 0`". For each source-destination pair your program will do the following:

- Perform a Breadth First Search (BFS) on the given source to assign a parent vertex to each vertex in the graph. The BFS algorithm will be discussed in class and is described in general terms below. The pseudo-code for BFS can be found on page 532 of the text.
- Use the results of BFS to print out the distance from the source vertex to the destination vertex, then use the parent pointers to print out a shortest path from source to destination. (See the algorithm Print-Path on p. 538 of the text.)

Examples

Input File:

```
6
1 2
1 3
2 4
2 5
2 6
3 4
4 5
5 6
0 0
1 5
3 6
2 3
4 4
0 0
```

Output File:

```
1: 2 3
2: 1 4 5 6
3: 1 4
4: 2 3 5
5: 2 4 6
6: 2 5

The distance from 1 to 5 is 2
A shortest path from 1 to 5 is: 1 -> 2 -> 5

The distance from 3 to 6 is 3
A shortest path from 3 to 6 is: 3 -> 1 -> 2 -> 6

The distance from 2 to 3 is 2
A shortest path from 2 to 3 is: 2 -> 1 -> 3

The distance from 4 to 4 is 0
A shortest path from 4 to 4 is: 4
```

If there is no path from source to destination (which may happen if the graph is disconnected), then your program should print a message to that effect. Note that there may be more than one shortest path. The particular path discovered by BFS depends on the order in which it steps through the vertices in your adjacency lists. This order is not specified in the pseudo-code for BFS. Thus your output may differ from the above on the very same input. The following example represents a disconnected graph.

Input File:

```
7
1 4
1 5
4 5
2 3
2 6
3 7
6 7
0 0
2 7
3 6
1 7
0 0
```

Output File:

```
1: 4 5
2: 3 6
3: 2 7
4: 1 5
5: 1 4
6: 2 7
7: 3 6

The distance from 2 to 7 is 2
A shortest path from 2 to 7 is: 2 -> 3 -> 7

The distance from 3 to 6 is 2
A shortest path from 3 to 6 is: 3 -> 2 -> 6

The distance from 1 to 7 is infinity
No path from 1 to 7 exists
```

Your program's operation can be broken down into two basic steps, corresponding to the two groups of input data.

1. Read and store the graph and print out its adjacency list representation.

2. Enter a loop which processes the second part of the input. Each iteration of the loop should read in one pair of vertices (source, destination), run BFS on the source vertex, print the distance to the destination vertex, then find and print the resulting shortest path, if it exists, or print a message if no path from source to destination exists.

What is Breadth First Search? Given a graph G and a vertex s, called the *source* vertex, BFS systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance from s to all such reachable vertices. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the unique path in the breadth-first tree from s to v is a shortest path in G from s to v. Breadth First Search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier; i.e. the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k+1$. To keep track of its progress and to construct the breadth-first tree, BFS requires that each vertex v in G possess the following attributes: a color $color[v]$ which may be white, gray, or black; a distance $d[v]$ which is the distance from v to the source s; and a parent (or predecessor) $p[v]$ which points to the parent of v in the breadth-first tree. At any point during the execution of BFS, the white vertices are those which are as yet undiscovered, black vertices are discovered, and the gray vertices form the frontier between discovered and undiscovered vertices. BFS uses a FIFO queue to manage the set of gray vertices. You will use your List ADT from pa1 to implement both this FIFO queue, and the adjacency lists which represent the graph itself.

Your Graph ADT will be implemented in a file called `Graph.java` which will define a class called Graph. Without going any further into the details of BFS, we can see a need for the following fields in your Graph class:

- An array of Lists whose i^{th} element contains the neighbors of vertex i.
- An array of Strings (or ints) whose i^{th} element is the color (white, gray, black) of vertex i.
- An array of ints whose i^{th} element is the parent of vertex i.
- An array of ints whose i^{th} element is the distance from the source to vertex i.

You should also include fields which store the number of vertices (called the *order* of the graph), the number of edges (called the *size* of the graph), and the vertex that was most recently used as a source for BFS(). An acceptable design option would be to include a private inner class called Vertex in your Graph class (much like the inner Node class in List.) The Vertex class could then encapsulate the color, parent, and distance-from-source for each vertex. The last three arrays mentioned above would then be replaced by a single Vertex array. It is recommended that all arrays be of length $n+1$, where n is the number of vertices in the graph, and that only indices 1 through n be used. This is so that array indices can be identified with vertex labels.

Your Graph class is required to include the following methods:

```
// Constructor
Graph(int n) // creates a graph with n vertices and no edges

// Access functions
int getParent(int u) // Returns the parent of vertex u
int getDist(int u) // Returns the distance from source to u
int getLastSource() // Returns the most recent source for BFS()
String getPath(int s, int u) // Returns string representation of shortest
                             // path from s to u
// Manipulation procedures
void addEdge(int u, int v) // Inserts new edge joining vertex u to vertex v
void BFS(int s) // Runs the BFS algorithm on source vertex s, and sets
                // distance and parent fields for each vertex in this graph

// Other methods
public String toString() // Returns adjacency list representation as a string
public static void main(String[] args) // Tests Graph ADT
```

Function `getPath(s,u)` is to be based on the algorithm Print-Path on p. 538 of the text, and will return a string representation of a shortest *s*-*u* path that is consistent with the output file format described above. Also `getPath(s,u)` has the precondition that the most recent call to `BFS()` will have been on source vertex *s*. Function `addEdge(u,v)` will place *u* in the adjacency list of *v*, and *v* in the adjacency list of *u*. Observe that any operation which takes a vertex label *u* as an input parameter is undefined if either $u < 1$ or $u > n$, and therefore all such operations have the precondition $1 \le u \le n$.

Submit the files README, Makefile, List.java, Graph.java, and FindPath.java to the assignment name pa3. Use peek or go to `/afs/cats.ucsc.edu/class/cmps101-pt.w06/pa3/foobar` (where `foobar` is your username) to check that the submission was complete. As always start early and ask questions.