Programming Assignment 4
Breadth First Search and Shortest Paths in Graphs
Due Tuesday May 21, 12:00 midnight

The purpose of this assignment is to understand and implement a Graph ADT and associated algorithms in C. The adjacency list representation of a graph consists of a sequence of lists, one list for each vertex, giving the neighbors of that vertex. For example, the graph

![Graph diagram]

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

In this assignment you will create a Graph ADT which represents an undirected simple graph as an array of neighbor-lists. Each vertex will be identified by an integer label from 1 to \( n \), where \( n \) is the number of vertices. Your program will use this Graph ADT to find shortest paths (i.e. paths with the fewest number of edges) between pairs of vertices.

Your main program, which will be called `findPath`, will take two command line arguments:

```
% findPath file1 file2
```

Here `file1` and `file2` are the input and output files respectively. The input file format is in two parts. The first part will begin with a line consisting a single integer \( n \) giving the number of vertices in the graph. The subsequent lines will each represent an edge in the graph by a pair of distinct numbers in the range 1 to \( n \), separated by a space. This first part of the input defines a 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.
The second part of the input 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 and a destination. For each such pair your program will do the following:

- Perform a Breadth First Search (BFS) to assign each vertex a parent vertex. 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 these parent vertices to print out a shortest path from the starting point to the destination, along with its length.

This second part of the input will also be terminated by the dummy line “0 0”. Continuing the previous example, the input lines

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

will result in the following lines being printed to the output file.

```
A shortest path from 1 to 5 of length 2 is: 1 -> 2 -> 5
A shortest path from 3 to 6 of length 3 is: 3 -> 1 -> 2 -> 6
A shortest path from 2 to 3 of length 2 is: 2 -> 1 -> 3
A shortest path from 4 to 4 of length 0 is: 4
```

If there is no path from the starting point to destination, 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.

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 (starting point, destination), run BFS on one of the vertices given, then find and print the resulting shortest path from starting point to destination, together with the length of the path.
What is Breadth First Search? Given a graph $G$ and a distinguished 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 $v$ corresponds to 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 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 in the algorithm’s execution, the white vertices are as yet undiscovered, black vertices are discovered, and the gray vertices form the frontier between discovered and undiscovered vertices. BFS uses a first-in first-out queue to manage the set of gray vertices.

Without going any further into the details of BFS, we can see a need for the following modules in your program design. Some of these are required in your project, while others are optional, and are so noted.

- A Vertex ADT with the fields label, distance, color, and parent. (Optional but recommended. Another option would be to build this into the Graph module as a private struct.)
- A FIFO Queue ADT used by BFS as mentioned above. (Optional since you can use your List ADT.)
- A List ADT to manage the adjacency lists. (Required.)
- A Graph ADT that uses all the other ADTs. (Required.)
- A main program called findPath which reads the input file, prints to the output file, and uses the services exported by Graph. (Required.)

Your Graph module is required to export a `GraphHndl` type and the following operations:

```c
GraphHndl NewGraph(int n);
void FreeGraph(GraphHndl* pGraph);
void addEdge(GraphHndl G, int u, int v);
void BFS(GraphHndl G, int s);
int parent(GraphHndl G, int u);
int distance(GraphHndl G, int u);
```

Function `NewGraph` returns a newly created Graph object with $n$ vertices and no edges, and of course `FreeGraph` de-allocates all memory associated with a Graph object. Function `addEdge` adds vertex $v$ to the adjacency list of $u$, and $u$ to the adjacency list of $v$. Function `BFS` implements the Breadth First Search algorithm on p. 532 of the text. Function `parent` returns the label of $u$’s parent in the BFS tree. This parent vertex is defined only after BFS has been run on a source vertex $s$. If `parent` is called before BFS, it should return some default value like zero. Function `distance` returns the distance from $u$ to the most recent source vertex $s$ on which BFS has been run. Always remember to adhere to the principle of modularity. Never let one module look inside the “black box” of another. For instance, the main program which is the client of Graph, should not be able to directly access the adjacency lists representing a Graph.

You may also implement additional Graph operations such as the following:

```c
Boolean isNull(GraphHndl G);
void makeNull(GraphHndl G);
GraphHndl graphCopy(GraphHndl G);
```
Function `isNull` returns true if its argument is a null graph, i.e. if \( G \) has no edges, `makeNull` deletes all edges from \( G \), making \( G \) a null graph, and `graphCopy` returns a new Graph object which is identical to its argument.

As usual you should submit a driver (e.g. `GraphDriver.c`) for each of your ADT modules, as well as the `.c` and `.h` files (`Graph.c`, and `Graph.h`). Also as usual, submit a README and Makefile. You may alter the following Makefile as necessary to fit your design.

```makefile
# Makefile for Graph ADT and related modules.
# make                makes findPath
# make GraphDriver    makes GraphDriver
# make ListDriver     makes ListDriver
# make VertexDriver   makes VertexDriver
# make clean          removes all object and executable files

findPath : findPath.o Graph.o List.o Vertex.o
  gcc -o findPath findPath.o Graph.o List.o Vertex.o

GraphDriver : GraphDriver.o Graph.o List.o Vertex.o
  gcc -o GraphDriver GraphDriver.o Graph.o List.o Vertex.o

ListDriver : ListDriver.o List.o
  gcc -o ListDriver ListDriver.o List.o

VertexDriver : VertexDriver.o Vertex.o
  gcc -o VertexDriver VertexDriver.o Vertex.o

findPath.o : findPath.c Graph.h
  gcc -c -ansi -Wall findPath.c

GraphDriver.o : GraphDriver.c Graph.h
  gcc -c -ansi -Wall GraphDriver.c

ListDriver.o : ListDriver.c List.h
  gcc -c -ansi -Wall ListDriver.c

VertexDriver.o : VertexDriver.c Vertex.h
  gcc -c -ansi -Wall VertexDriver.c

Graph.o : Graph.c Graph.h List.h Vertex.h
  gcc -c -ansi -Wall Graph.c

List.o : List.c List.h
  gcc -c -ansi -Wall List.c

Vertex.o : Vertex.c Vertex.h
  gcc -c -ansi -Wall Vertex.c

clean :
  rm findPath GraphDriver ListDriver VertexDriver \ findPath.o GraphDriver.o ListDriver.o VertexDriver.o \ Graph.o List.o Vertex.o
```