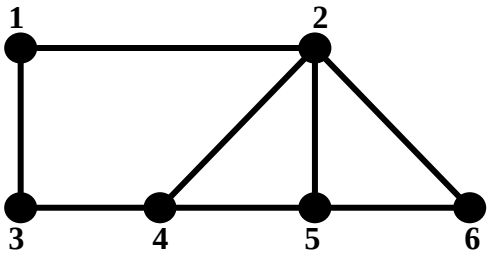
CMPS 101
Algorithms and Abstract Data Types
Programming Assignment 4

Breadth First Search and Shortest Paths in Graphs

The purpose of this assignment is to implement a Graph ADT and associated algorithms in C. This project will utilize your List ADT from pa2 so spend some time going over the grader's comments to make sure your List is working properly. Begin by reading the handout on Graph Theory, as well as appendices B.4, B.5 and sections 22.1, 22.2 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 that 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 using your Graph ADT will be called `FindPath`, and will take two command line arguments (here `%` denotes the unix prompt):

```
% FindPath input_file output_file
```

As in prior projects, you are to write a Makefile that creates the executable file, called FindPath in this case, and includes a clean utility.

File Formats

The input file will be in two parts. The first part will begin with a line consisting of a single integer *n* giving the number of vertices in the graph. Each subsequent line will represent an edge 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) from the given source vertex. This assigns a parent vertex (which may be nil) to every 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 in section 22.2 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 in section 22.2 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 1-5 path is: 1 2 5

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

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

The distance from 4 to 4 is 0
A shortest 4-4 path 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 joining a given pair of vertices. The particular path discovered by BFS depends on the order in which it steps through the vertices in each adjacency list. We adopt the convention in this project that vertices are always processed in sorted order, i.e. by increasing labels. The output of BFS is uniquely determined by this requirement. 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 2-7 path is: 2 3 7

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

The distance from 1 to 7 is infinity
No 1-7 path 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 that 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 (as in the above example).

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 BFS tree with root *s* that contains all vertices reachable from *s*. For any vertex *v* reachable from *s*, the unique path in the BFS 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 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 source *s* to vertex *v*; and a parent (or predecessor) *p*[*v*] that refers to the parent of *v* in the BFS tree. At any point during the execution of BFS, the white vertices are those that 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. Use your List ADT from pa2 to implement both this FIFO queue, and the adjacency lists representing the graph itself.

Your Graph ADT will be implemented in files `Graph.c` and `Graph.h`. `Graph.c` defines a struct called `GraphObj`, and `Graph.h` will define a type called `Graph` that points to this struct. (It would be a good idea at this point to re-read the handout *ADTs and Modules in Java and Ansi C*, paying special attention to the section on implementing ADTs in C, as well as the handout *Some Additional Remarks on ADTs and*

Modules in Ansi C.) Without going any further into the details of BFS, we can see a need for the following fields in your struct `GraphObj`:

- An array of Lists whose i^{th} element contains the neighbors of vertex *i*.
- An array of ints (or chars, or strings) 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 (most recent) source to vertex *i*.

You should also include fields storing the number of vertices (called the *order* of the graph), the number of edges (called the *size* of the graph), and the label of the vertex that was most recently used as source for BFS. 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 directly identified with vertex labels.

Your Graph ADT is required to export the following operations through the file `Graph.h`:

```
/*** Constructors-Destructors ***/
Graph newGraph(int n);
void freeGraph(Graph* pG);

/*** Access functions ***/
int getOrder(Graph G);
int getSize(Graph G);
int getSource(Graph G);
int getParent(Graph G, int u);
int getDist(Graph G, int u);
void getPath(List L, Graph G, int u);

/*** Manipulation procedures ***/
void makeNull(Graph G);
void addEdge(Graph G, int u, int v);
void addArc(Graph G, int u, int v);
void BFS(Graph G, int s);


/*** Other operations ***/
void printGraph(FILE* out, Graph G);
```

In addition to the above prototypes `Graph.h` will define the type `Graph` as well as `#define` constant macros `INF` and `NIL` that represent infinity and an undefined vertex label, respectively. For the purpose of implementing BFS, any negative `int` value is an adequate choice for `INF`, and any non-positive `int` can stand in for `NIL`, since all valid vertex labels will be positive integers. `INF` and `NIL` should of course be different integers.

Function `newGraph()` returns a `Graph` pointing to a newly created `GraphObj` representing a graph having *n* vertices and no edges. Function `freeGraph()` frees all dynamic memory associated with the `Graph *pG`, then sets the handle `*pG` to `NULL`. Functions `getOrder()` and `getSize()` return the corresponding field values, and `getSource()` returns the source vertex most recently used in function `BFS()`, or `NIL` if `BFS()` has not yet been called. Function `getParent()` will return the parent of vertex *u* in the Breadth-First tree created by `BFS()`, or `NIL` if `BFS()` has not yet been called. Function `getDist()` returns the distance from the most recent BFS source to vertex *u*, or `INF` if `BFS()` has not

yet been called. Function `getPath()` appends to the List *L* the vertices of a shortest path in *G* from source to *u*, or appends to *L* the value `NIL` if no such path exists. `getPath()` has the precondition `getSource(G)!=NIL`, so `BFS()` must be called before `getPath()`. Functions `getParent()`, `getDist()` and `getPath()` all have the precondition $1 \leq u \leq \text{getOrder}(G)$. Function `makeNull()` deletes all edges of *G*, restoring it to its original (no edge) state. (This is called a *null graph* in graph theory literature). `addEdge()` inserts a new edge joining *u* to *v*, i.e. *u* is added to the adjacency List of *v*, and *v* to the adjacency List of *u*. Your program is required to maintain these lists in sorted order by increasing labels. `addArc()` inserts a new directed edge from *u* to *v*, i.e. *v* is added to the adjacency List of *u* (but not *u* to the adjacency List of *v*). Both `addEdge()` and `addArc()` have the precondition that their two `int` arguments must lie in the range 1 to `getOrder(G)`. Function `BFS()` runs the BFS algorithm on the Graph *G* with source *s*, setting the color, distance, parent, and source fields of *G* accordingly. Finally, function `printGraph()` prints the adjacency list representation of *G* to the file pointed to by `out`. The format of this representation should match the above examples, so all that is required by the client is a single call to `printGraph()`.

As in all ADT modules written in C, you must include a test client called `GraphTest.c` that tests your Graph operations in isolation. Observe that since the Graph ADT includes an operation having a List argument (namely `getPath()`), any client of Graph is also a client of List. For this reason the file `Graph.h` should `#include` the header `List.h`. (See the handout *C Header File Guidelines* for standard policies on `.h` files.)

You will submit eight files in all for this project:

```
List.c
List.h
Graph.c
Graph.h
GraphTest.c
FindPath.c
Makefile
README.
```

A makefile is included on the website that you may change as you see fit. You will also find a file called `GraphClient.c` that computes a few graph theoretic quantities using BFS. Do not turn in this file, but use it in your own tests if you like. It should be considered to be a weak test of our Graph ADT and does not take the place of your own `GraphTest.c`.