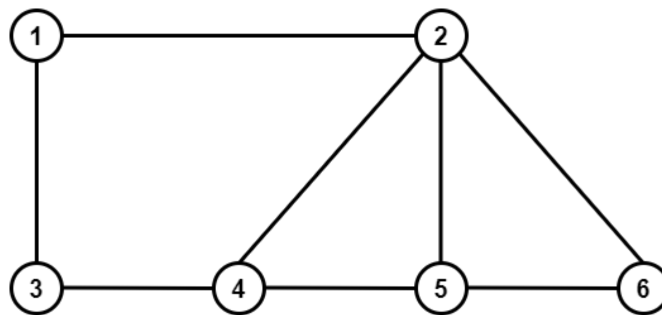
CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 2

Breadth First Search and Shortest Paths
The purpose of this assignment is to implement a Graph ADT in C, along with the Breadth First Search algorithm for finding shortest paths in a graph. This project will utilize your List ADT from pa1. Begin by reading the handout on Graph Algorithms, as well as appendices B.4, B.5 and sections 20.1, 20.2 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)
```

Your Graph ADT will use this method for representing a graph. Each vertex will be identified with an integer label in the range 1 to n, where n is the number of vertices in the graph. The client program in this project will be called `FindPath.c`, and will use the Graph ADT to find shortest paths (i.e. paths with fewest edges) between pairs of vertices. It will take two command line arguments, as follows.

```
$ FindPath <input file> <output file>
```

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 (source) and an ending point (destination). The second part of the input 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 (also called a *predecessor*, 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 20.2 of the text, and is also presented at Examples/Pseudo-Code/GraphAlgorithms on the class webpage.

- Use the results of BFS to print out the distance from the source vertex to the destination vertex, then use the predecessors to recursively print out a shortest path from source to destination. See the algorithm Print-Path in section 20.2 of the text, and also at Examples/Pseudo-Code/GraphAlgorithms.

Examples

<u>Input File:</u>

```
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
```

<u>Output File:</u>

```
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 will 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 vertex labels. The output of BFS is uniquely determined by this requirement. Therefore your Graph ADT should maintain the adjacency lists in sorted order. 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 that 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 finished, and the gray vertices are discovered, but not all of their neighbors have been discovered. The gray vertices thus form the frontier between undiscovered and finished vertices. BFS uses a FIFO queue to manage the set of gray vertices. Use your List ADT from pa1 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`. File `Graph.c` will define a struct called `GraphObj`, and `Graph.h` will define a type called `Graph` that is a pointer to this struct. (It would be a good idea at this point to re-read the handout ADTs-in-C.pdf.) 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 undirected edges, and the number of directed edges (also called arcs), 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 will export the following operations through the file `Graph.h`:

```
// constructors - destructors -------------------------------------------------

// newGraph()
// Returns a Graph having n vertices and no edges.
Graph newGraph(int n);

// freeGraph()
// Frees all dynamic memory associated with Graph *pG and sets *pG to NULL.
void freeGraph(Graph* pG);


// access functions -----------------------------------------------------------

// getOrder()
// Returns the number of vertices in G.
int getOrder(Graph G);

// getNumEdges()
// Returns the number of edges in G.
int getNumEdges(Graph G);

// getNumArcs()
// Returns the number of Arcs in G.
int getNumArcs(Graph G);

// getSource()
// Returns the source vertex in the most recent call to BFS(), or NIL if
// BFS() has not yet been called.
int getSource(Graph G);

// getParent
// Returns the parent of vertex u in the most recently constructed BFS tree
// or returns NIL if BFS() has not yet been called.
// Pre: 1 <= u <= getOrder(G)
int getParent(Graph G, int u);
```

```
// getDist()
// Returns the distance from the source vertex to u if BFS() has been called,
// otherwise returns INF.
// Pre: 1 <= u <= getOrder(G)
int getDist(Graph G, int u);

// getPath()
// If vertex u is reachable from the source, appends the vertices of a shortest
// source-u path to List L. Otherwise, appends NIL to L.
// Pre: 1 <= u <= getOrder(G), getSource(G) != NIL
void getPath(List L, Graph G, int u);


// manipulation procedures -----------------------------------------------------

// makeNull()
// Resets G to its initial state.
void makeNull(Graph G);

// addEdge()
// Creates an undirected edge joining vertex u to vertex v.
// Pre: 1 <= u <= getOrder(G), 1 <= v <= getOrder(G)
void addEdge(Graph G, int u, int v);

// addArc()
// Creates a directed edge joining vertex u to vertex v.
// Pre: 1 <= u <= getOrder(G), 1 <= v <= getOrder(G)
void addArc(Graph G, int u, int v);

// BFS()
// Runs the Breadth First Search algorithm on G with source vertex s.
void BFS(Graph G, int s);


// other functions -------------------------------------------------------------

// printGraph()
// Prints the adjacency list representation of G to FILE* out.
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 heap memory associated with the `Graph *pG`, then sets the handle `*pG` to `NULL`. Functions `getOrder()`, `getNumEdges()` and `getNumArcs()` 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 BFS 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()` is called. 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 (called a *null graph* in graph theory literature). Function `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 vertex labels. Function `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. It also has the precondition that the `int` argument s lies in the range 1 to `getOrder(G)`. 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 generally accepted policies on using `.h` files.) As in pa1, you will write a Makefile that creates the executable binary called FindPath. Include a clean utility in your Makefile that removes all binary files. A Makefile is included on the website that you may change as you see fit. Also included is a random input file generator which you may use for testing called RandomInput2.py

Thus, you will submit eight files in all:

`List.c`	written by you
`List.h`	written by you
`Graph.c`	written by you
`Graph.h`	written by you
`GraphTest.c`	written by you
`FindPath.c`	written by you
`Makefile`	provided, alter as needed
`README.md`	a list of files submitted, and any notes to the grader

You will also find a file called `GraphClient.c` in /Examples/pa2 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`.

Please start this project early and get help from myself, Course Tutors and TAs as needed.