CMPS 101
Algorithms and Abstract Data Types
**Spring 2008**

Programming Assignment 4
Breadth First Search and Shortest Paths in Graphs
Due Tuesday May 27, 10:00 pm

The purpose of this assignment is to implement a Graph ADT and associated algorithms in C. This project will utilize your List ADT from programming assignment 2, 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
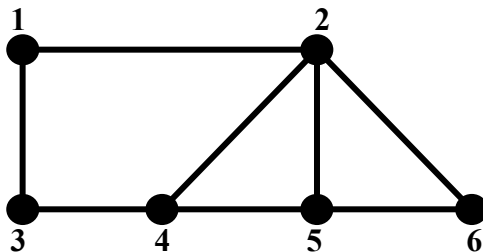
has adjacency list representation

```
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 ListRefs. 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 pa2, you are to write a Makefile which creates the executable file FindPath. One possible Makefile, which you may alter as you see fit, is included at the end of this document. 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.

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 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 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 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. 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:                Output File:
7                          1:  4  5
1  4                       2:  3  6
1  5                       3:  2  7
4  5                       4:  1  5
2  3                       5:  1  4
2  6                       6:  2  7
3  7                       7:  3  6
6  7
0  0                       The distance from 2 to 7 is 2
2  7                       A shortest 2-7 path is: 2 3 7
3  6
1  7                       The distance from 3 to 6 is 2
0  0                       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 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 source *s* to vertex *v*; 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 pa2 to implement both this FIFO queue, and the adjacency lists which represent the graph itself.

Your Graph ADT will be implemented in files `Graph.c` and `Graph.h`. `Graph.c` will define a struct called Graph, and `Graph.h` will define a type called GraphRef which points to this struct. (It would be a good idea at this point to re-read the handout entitled "ADTs and Modules in Java and Ansi C", paying special attention to the section on implementing ADTs in C, as well as the handout entitled "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 Graph struct:

- 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 which store 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 identified with vertex labels.

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

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

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

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


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

In addition to the above prototypes, `Graph.h` will define the type `GraphRef`, as well as `#define` constant macros `INF` and `NIL`, representing infinity and a blank 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.

Function `newGraph()` returns a `GraphRef` pointing to a newly created `Graph` struct representing the null graph on *n* vertices, i.e. a graph having *n* vertices and no edges. Function `freeGraph()` frees all dynamic memory associated with the `GraphRef *pG`, then sets *pG to NULL. Functions `getOrder()`, `getSize()` will return the corresponding field values, and `getSource()` will return 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 that `BFS()` must be called before `getPath()`. Functions `getParent()`, `getDist()` and `getPath()` all have the precondition $1 \le u \le \text{getOrder}(G)$. Function `makeNull()` deletes all edges of *G*, restoring it to its original state, i.e. a null graph on *n* vertices. `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*. `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 Breadth First Search algorithm on the Graph *G* with source *s*, setting the color, distance, and parent 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 exercises your Graph operations in isolation. Observe that since the Graph ADT includes an operation that has a List argument (namely `getPath()`), any client of Graph is also a client of List, and must therefore `#include` the file `List.h`. This goes for both `GraphTest.c` and `FindPath.c`. Note however that `List.h` must be `#included`'ed before `Graph.h`, otherwise the compiler will choke on the prototype for `getPath()`. This is just one of the subtleties one faces when programming in Ansi C (as opposed to Java). You will submit nine files in all for this project: `List.c`, `List.h`, `ListTest.c`, `Graph.c`, `Graph.h`, `GraphTest.c`, `FindPath.c`, `Makefile`, and `README`. You may use the following Makefile as a starting point. Note that the command `make` (or `gmake`) alone will compile the first target in the Makefile, along with all its dependent targets, while `make target` will compile any other target.

```
#    Makefile for Programming Assignment 4

FindPath: FindPath.o Graph.o List.o
     gcc -o FindPath FindPath.o Graph.o List.o

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

GraphTest: GraphTest.o Graph.o List.o
     gcc -o GraphTest GraphTest.o Graph.o List.o

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

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

ListTest: ListTest.o List.o
     gcc -o ListTest ListTest.o List.o

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

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

clean :
     rm -f FindPath GraphTest ListTest FindPath.o \
          GraphTest.o ListTest.o Graph.o List.o
```