# CMPS 101 - Programming assignment 5
# Giving Directions

Due 11:55pm Friday 6/6/14, no extensions.

May 22, 2014

## 1 Introduction

Google maps and other programs have a valuable feature that prints directions on how to get from one place to another. You are to implement this kind of service on simple graphs using breadth-First-Search.

For this assignment you are encouraged to work in groups of two. Partnerships must rotate – you may not use the same partner as a previous programming assignment. Both student's are responsible for ensuring that both partners completely understand the code and constructs used. Every file should begin with a block comment indicating who the partners are (names and UCSC accounts) and *any* help received on that part of the program. Significant outside help must also be acknowledged in the README file. Only one partner should submit the zip file described below, the other partner should watch the submission to ensure it is done correctly and on time. The partner not submitting the program should instead just submit a copy of the partnership's README file (*not* zip'ed).

The goals of this assignment include:

- Reuse of the list ADT
- Implementing a graph and a graph algoritm
- Gain practice with BFS

## 2 Program Specifications

I encourage you to base your list ADT on your solution from a previous program. However, you *must* credit any previous partner(s).

Your program is to read in and process a file (given as the first command line argument, use `argc` and `argv`. and print a set of directions the standard output. The input file will consist of a number of lines formatted as follows:

- The first line will contain two integers. The first is `numVerts`, the number of vertices (locations) in the graph. Vertices will be numbered starting at 0, so if `numVerts` is 7, the vertices will be numbered 0 to 6. The second integer is `numQueries`, the number of path queries in the file. This is provided simply to make reading the queries easier.
- The next `numVerts` lines define the graph. Each of the next number-of-vertices lines is an adjacency list. It will start with the vertex number that edges leave, and then contain a (possibly empty) list of neighboring vertices (that edges go to). Thus if one of these lines is:

  6 7 1 4 2

  It indicates that there are edges going from vertex 6 to 7, from 6 to 1, from 6 to 4, and from 6 to 2. Each vertex will have one line, and the lines will appear in vertex order. *Note: if you are not comfortable reading variable-length input lines, experiment with it and/or get help with this right away!*

- The remaining `numQueries` lines in the file each contain a direction query consisting of the start vertex number and the destination vertex number.

The input graph will be a *directed* graph, so that if there is an edge from vertex 3 to vertex 7, there may, or may not, be an edge from vertex 7 back to vertex 3.

Your program should read in the entire file storing the graph. Then for each query line it it should read the start vertex and destination vertex. It should then perform a Breadth First Search from the start vertex, and build the shortest path tree. Finally, it should print the distance from the start vertex to the destination and the shortest path from the start vertex to the destination discovered by your BFS. If there is no path from the start vertex to the destination, then instead of printing a path it should print an easily understood message indicating that no path exists. Each query should be processed in this way, first reading query, then performing another BFS (after appropriate initialization), and then printing the directions. After the `numQueries` queries have all been processed, your program should clean up and exit. Separate the directions from different queries with a blank line.

Example; if the file was: (hint: draw the directed graph corresponding to these adjacency lists)

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

The output should be something like:

```
To get from 0 to 4 requires 2 edges,  follow the path 0, 2, 4.

I am sorry, there is no way to get from 4 to 0 in this graph.
```

## 3 Program Design – top down description

Your program will probably make heavy use of your list ADT module. The uses for lists of vertex numbers include: adjacency lists, the queue used by BFS, and the stack used to reconstruct the path. You should implement the graph module as a separate directed graph ADT.

There are design choices for how to implement the graph module and what its responsibilities should be. In any case, the graph ADT should have operations like:

- `newGraph(int n)` – creates a new graph with $n$ vertices numbered 0 to $n-1$ and no edges, and returns a handle to it.
- `addEdge(Graph G, int from, int to)` – adds the edge (from, to) to the graph.
- `freeGraph(Graph G)` – frees the storage used by a no longer needed Graph ADT object.

One design choice is s a heavy-weight graph ADT that also implements the BFS functionality. This is the approach I recommend. The BFS data structures will then be part of the graph ADT. In this case, the graph ADT will also have operations like:

- `doBFS(Graph G, int source)` – performs BFS and updates the graph's parent and distance arrays. This will erase the results of any previous BFS.
- `getDistance(Graph G, int destination)` – return the number of edges in the shortest path from the last BFS source to the destination. You can return `-1` or the `maxint` constant if there is no path from the source to the destination (but this choice must be clearly documented!)

- `getPathTo(Graph G, int destination)` – return a List containing the path from the last doBFS source to the destination, or a null pointer if no such path exists.

Note that the graph ADT creates the path-list on behalf of the client, and it is the client's responsibility to free up that list's storage.

Another choice is to have the BFS performed by a client of the graph, so the BFS data structures will belong to the client. In this case, the graph should have an operation like:

- `getAdjacencyList(Graph G, int vertex)` – return the adjacency list for the vertex

One disadvantage of this method is that the adjacency lists are exported to the client, so the ADT risks having the client destroy its data integrity.

You may add additional operations and/or modify the above to suit your own design choices. In particular, `printGraph` and `printBFSTree` will help with debugging your graph module. Each operation should be fully specified with pre- and post- conditions in the `graph.h` file, and there should be a driver program to test them out. Note that both the graph module and the main program will both need to know about the List type used to store paths. It will be most convenient if the list module has a current location concept (as in the first program specifications) that allows clients to step through the list.

The program will then work by

1. reading the first line and remembering the number of vertices and number of queries,
2. reading the next number-of-vertex lines while constructing the graph,
3. reading each start-destination query line and:
   (a) performing BFS on the graph with the source being the start vertex
   (b) using the calculated parent pointers from BFS, recover a shortest path from the start vertex to the destination.
   (c) printing this path

   before reading the next start-destination query line.

## 4 Implementation Details - bottom up

You should implement your program with a number of modules using the ADT methodology. Get started early and implement the modules in a bottom-up way.

1. At the bottom level, you will need lists of vertices. Since the vertices are numbered, a list-of-integers module will do the job. You will want a list module with the concept of current location so you can easily write loops that step through the list.
2. The graph should be implemented as an array of vertices, where each slot in the array is an adjacency list. Decide if you want to implement the empty adjacency list by the empty list or a null handle (I recommend using an empty list). In order to do BFS, the graph will also need an array indexed by vertices to hold the colors (white, grey, black), an array indexed by vertices to hold the distances, and an array of vertices to hold the parent vertex-IDs (which are integers). These arrays used by BFS should be re-initialized each time `doBFS` is called.
3. Once the BFS is performed and the parent values calculated, you will need to produce the path from the source to the destination. This requires that you reverse the list of parents going from the destination back to the source. One way to do this uses the List ADT as in the following pseudo-code:

```
ListHandle path = newLIst();
nextUp = destination
while nextUp != source
    insertBeforeFirst(path, nextUp);
    nextUp = parentArray[nextUp];
insertBeforeFirst(path, source);
```

Now reading `path` from front to back contains the shortest route from the source to the destination. Of course, you need to check that a path to the destination exists first.

## 5 Submission

As with the other programs, zip up all of your files (including drivers, Makefile, and README) into a file named *lastNameFirstInitial*`Prog4.zip` and submit it through E-commons. By all your files, I mean: the main program for the assignment, your Makefile, and your README, as well as the .h files, .c files, and drivers for all of your modules that are needed to run your main program. Your Makefile should compile all of the .c files and create executables for the main program and each of the drivers. *If you don't know how to do this, get help in section!*

As always, exported functions should be described (complete with pre- and post- conditions) in the appropriate `.h` file. I expect each submission will contain at least a main program, a list module, and possibly a graph module. Feel free to break the problem into additional modules if it makes things easer.

As with previous assignments, you will need to make sure that the programs compile and run correctly on the UCSC unix systems. You will lose points for makefile and/or compile issues, and should not expect that graders will take much time trying to fix them. If the program does not compile, graders may choose to grade the code on non-execution aspects and this will significantly cost your grade.

### 5.1 Grading Guide

15 points maximum:

- 1 point for correct submission with all files (including driver programs, README, and Makefile).
- 1 point for using separate ADT Files with information hiding and good descriptions of the exported functions including pre- and post- conditions clearly specified
- 1 point for well organized, commented, readable code and generally good style,
- 2 points for *a good, self-contained, easy to use* Makefile.
- 3 points for correctly computing the shortest path distances.
- 6 points for correctly outputting the shortest paths.
- 1 point for including a graph module separate from (but using) the list module, and separate from the main program.

Bonus points: These can be used to make-up lost points, but cannot increase your assignment total above 10.

- 1 point for good C memory management (each module frees the memory it ”owns”).

## 6 Notes and Hints:

- The end of the quarter is a busy time for everyone, and there is another written assignment coming, so **get started early**.
- Visualize the program top-down to understand what needs to be done, but implement bottom-up, using a driver to check that each small part of the implementation is working well before moving on to the next small chunk. If in any doubt, *verify that your list ADT is working ASAP*.
- Write the makefile early. This will help ensure you are testing with the current versions of each module.
- One of the first chunks to write for each module is a print function that will help check correctness and aid debugging.
- Although the graph module creates a linked list containing the path, it should be the client's responsibility to free this list since only the client knows when they are done with it.
- Test your programs on both small and large input files. How long does it take to run on really big graphs (with tens of thousands of vertices)?