

## Program #3 – Efficient Bike Paths

**Due midnight, Wednesday November 25**

The purpose of this assignment is to understand and implement a graph ADT and associated algorithms.

### 1 Introduction

The town of Los Vertices has a number of bicycle paths connecting its neighborhoods. Your job is to write a program that helps cyclists navigate from one neighborhood to another as efficiently as possible.

The input to your program comes in two parts: first there is a map of Los Vertices indicating the bike paths between neighborhoods. You may assume that there are at most 99 neighborhoods, and the neighborhoods will be identified by (different) integers from 1 to the number of neighborhoods. Your program should read in this map and store it as an undirected graph in adjacency list form. Following the map will be a set of route requests giving a starting neighborhood and a desired destination. After reading in each route request, your program should find and print out a shortest (fewest number of edges) path from the starting neighborhood to the destination.

The suggested implementation is to use a breadth-first search for each navigation request to store a shortest path tree in the graph, and then use that shortest path tree to print out the needed path.

### 2 Technical details

Each neighborhood (vertex) is identified by a single integer. Neighborhoods will be numbered consecutively starting with 1. The first line of the map contains an integer  $n$  giving the number of neighborhoods in the map. The next lines each represent a bikepath, listing the two integers for the neighborhoods connected by the path, and a string (of at most 20 characters without spaces) for the path's name. These names are used only for the extra credit, so you should start by ignoring them. After the map will be a dummy line containing "0 0 0" – this dummy line marks the end of the map and is not a bike path. At this point you should print out the adjacency list representation of the map so we can check that your program is reading in the graph correctly.

After the dummy line, there will be a series of navigation requests. Each navigation request is a line with two integers, the first giving the starting neighborhood and the second the destination. For each navigation request you should:

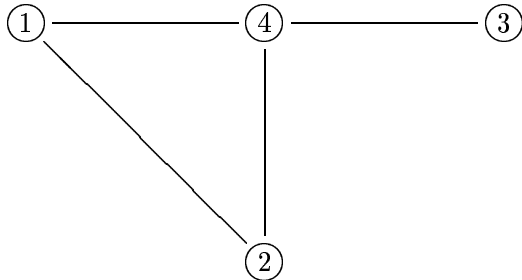
1. perform a breadth first search to give each neighborhood a "parent" neighborhood, and then
2. use these "parent" neighborhoods to print out the shortest (in terms of number of bikepaths) path from the start neighborhood to the destination.

The last line in the file is another dummy line, containing "0 0"

Thus one map of a 4-neighborhood district would be input like the following:

```
4
1 2 Meadow.drive
4 1 Falcon's.peak
2 4 Sand.n.surf
3 4 No.Joggers
0 0 0
```

This corresponds to the following graph



The adjacency list representation of the graph might be printed as:

```
1: 2, 4,
2: 1, 4,
3: 4,
4: 1, 2, 3,
```

Once it has read in the graph and output the adjacency list your program should be ready to accept routing requests. Each request will be a pair of neighborhood numbers on a single line. If the request is from neighborhood  $u$  to neighborhood  $v$  then the input line will be “ $u v$ ” and the program should print a shortest path that goes from neighborhood  $u$  to neighborhood  $v$ .

For example, using the above graph, the request line “2 3” would cause something like:

To get from neighborhood 2 to neighborhood 3, go:

```
2 to 4 to 3.
```

to be printed.

You may assume that the graph is connected, so some path will always exist. You might want to print something clever if the request is how to get from a neighborhood to itself.

### 3 Implementation Strategy

Your program’s activity can be broken down into 2 major steps.

- Read and store the graph (map of the neighborhoods) and output the adjacency list representation of the graph.
- Enter a loop processing navigation requests. Each iteration of the loop should read in one request, run Breadth First Search starting from one of the vertices given, and then find and print the resulting shortest path to the destination.

I suggest that you design your program using the following modules.

- A list ADT for the adjacency lists.

- A graph module using the list ADT.
- A main program that reads the input, prints the output, and uses the graph module.

Most of the real work in this program will be in the graph module. In addition to the graph's adjacency lists, the graph should also have a queue and arrays for  $\pi$ ,  $d$ , and a color that will be used by the Breadth-first search.

The graph module should export the following operations:

- `InitGraph(&graph,nodes)` – initialize the graph  $G$  to be the  $n$ -node graph with no edges. The nodes are numbered 1 to  $n$ .
- `AddEdge(graph,u,v)` – add the undirected edge from node  $u$  to node  $v$  in graph  $G$ , putting it on adjacency lists for both  $u$  and  $v$ .
- `BFS(graph, u)` – This operation should perform a BFS in the graph rooted at  $u$ , to set the parent values for each vertex. You may need to clean up stuff stored by the previous BFS before the new BFS can be started (i.e. resetting parents and colors).
- `ParentOf(graph,u)` – returns the parent of node  $u$  set by the last BFS.

The main program will use the `ParentOf()` function to recover and print the shortest path.

You will probably also want a `DumpBFS` procedure to print out all the information, including the node colors, the queue, and the  $\pi$  and  $d$  values, to help you debug your program. As always, the graph ADT should also have a `DeleteGraph` destructor operation that cleans up any storage allocated for the graph. Note: the `graph` parameter in the operations is optional, see the extra credit section. For the basic assignment your Graph Module need only deal with a single graph.

To get your graph module working, start small. First write a driver which initializes a graph with a few vertices and then prints it out. You are to store the graph in adjacency list form, the recommended representation is as an array of vertices, where each vertex has its own color, parent, adjacency list, etc.

Next work on reading the input from the main program and getting the BFS routine working. The BFS pseudo-code is on page 470 of the text. You will need a queue of vertices to run the BFS. Either build a queue ADT or use an `intlist` for the queue. (Note that the queue ADT should have its own `.h.c` module)

After your BFS is working, work on implementing the `FindPath` routine which recovers paths from the  $\pi$  array.

I expect that many student's will implement their graph modules to support just one graph. Therefore, the "graph" you are working on is understood and need not be a parameter to the graph operations. However, the following is worth extra credit:

- 5 points for implementing the graph ADT so that the client can operate with several graphs at once.
- 5 points for giving the name for each path as they are printed in route, e.g.:

To get from neighborhood 2 to neighborhood 3, take:  
 Sand.n.surf from 2 to 4, and No.Joggers from 4 to 3.

These names should be stored and recovered from your graph ADT. Therefore graph ADT operations will have to be added and/or modified.

The late deadline is midnight Monday November 30. In addition to the normal 30 point late penalty, late programs will NOT be eligible for extra credit.