University of Illinois at Urbana-Champaign
Department of Computer Science

MP9 : Introduction to the `Graph` class

CS 225 Data Structures
Spring Semester, 1999

Handed out: Sunday, April 18, 1999
Due date: Saturday, April 24, 1999 at 5:00 PM

1 Introduction

In this MP, you will get to work with the graph class that we discussed in lecture and whose code we went over in section. You will implement two graph algorithms using the graph interface, and also add two member functions to the graph class itself.

2 The given files

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp9          (MP code)
~cs225/src/mp9/test     (test files)
```

Take some time to carefully go over all the function definitions in the graph.C and infograph.C files.

3 Your Assignment

3.1 Part 1 - using the interface

For the first part of the assignment, you are going to implement the Breadth First Search (BFS) and Depth-First Search (DFS) algorithms. You will do this from the point of view of the user. That is, rather than change the actual implementation of the class itself, you will instead use the Graph interface to write the BFS and DFS functions. In the file `graphalgs.h`, you will see function headers for 3 functions that you need to write. The specification for these functions is as follows :

- The BFS function. This function takes two parameters - one is the graph to perform the BFS on and the other is a *start* vertex. You are allowed to use a queue to implement your BFS. A queue class has been provided. See files *queueal.h* and *queueal.C*. The BFS function must print out the node-information for each node in BFS order.
- The DFS function. This function again takes two parameters - the graph and a *start* vertex. This will be a helper function that starts off the actual recursive DFS.
- The RecursiveDFS function. This is the actual recursive DFS function that is called by the helper above. It will not return any value, but must print out the node-information for each node in DFS order.

You will find it very helpful to look at the `graph.h` and `graphalgs.*` files provided in the mp9 directory, as well as the `Graph` class interface notes posted earlier. The `graphalgs.C` file has an implementation of the `Print` function already written, and you can take hints from that when writing your functions. Also, look at the pseudocode for these algorithms in chapter 12 of the text. The interface is designed so that you can easily translate from the pseudocode of a graph algorithm to an actual coding of it using this interface. Note that all these functions invove the `Infograph` class rather than the regular `Graph` class, because we are associating data with the vertices.

It is okay to assume for the purposes of this MP that the traversals will always reach all vertices no matter where you start. So, you don't need to worry about starting over when you do a search like we talked about in lecture. There will be only a breadth-first or depth-first spanning tree, not a spanning forest.

3.2 Part 2 - adding to the class itself

The second part of the assignment will involve adding a function to the `Graph` class itself, a function called `ReverseEdge()`. This function takes an edge for a parameter and reverses that edge – i.e. the source vertex for the edge becomes the target vertex, and vice-versa. For the implementation to work right, there are other things that need to be changed as well, but abstractly, that is what needs to be done.

There is a lot of code for this function, but it is not so much "hard to figure out"-style code as it is "there is a lot to do and the variable names are long"-style code. Below is the basic outline for reversing a directed graph edge in our implementation

```
1) extract the edge from its source vertex list
    and its target vertex list
2) switch the role of target and source for this vertex
3) insert into new source vertex and target vertex lists
```

The difficulty comes due to two things – first, the lists are doubly linked, which means an insertion needs four pointer changes rather than two; and second, a vertex's edge list could be empty after removal or become non-empty due to insertion, meaning that one or more vertex node fields need to be changed as well.

There are a few other details, but if you understand the implementation, things will be clear. Note that YOU ARE NOT ALLOWED TO USE ALREADY WRITTEN HELPER FUNCTIONS!!!! For example, you cannot implement this function by calling InsertEdge or any of its helper functions. You CAN use the code from those functions if you find copying and pasting useful, because at least in that circumstance you have to at least understand the implementation well enough to know what to copy. But if you use the interface, you don't need to understand the implementation, and we do want you to wrap your brain around the implementation for a bit.

It is okay to assume for the purposes of this MP that your graph will be directed. That is, your `ReverseEdge` function does not have to worry about the undirected edge case. It turns out that the code for that case is actually quite small when compared to the directed edge case that you need to write, but since you have enough to do with just the directed edge case, it is okay to ignore the undirected edge part of things.

As with the Part 1 functions, we have already given you the function header in the interface file (here, the `graph.h` file). It is located in the "Graph structure alteration" section of `graph.h`. In addition, we have indicated a space at the top of `graph.C` where we want you to add this function (so that the grader doesn't need to scroll through the code to find your functions).

There are three more rules to follow here, rules which will help you match our output:

1. Whenever you traverse down an edge list, traverse from the first vertex to the last vertex (i.e. follow the "next" pointers, and not the "previous" pointers). This will help you match our output in the traversal functions in Part 1.

2. Whenever you insert an edge in an edge list, it should be inserted as the last edge in the edge list. This is important when you are re-inserting the reversed edge in part 2. The code we have given you already does the same thing – always inserts at the end – so looking at code where we insert into a doubly-linked list can help you.

3. Whenever you remove the first node from an edge list, the pointer to that first node should point to the second node and make that the new first node. This is important when you are extracting the edge from its source and target edge lists in part 2. At times, this could be the first vertex in the edge list, which means that its source vertex's departingEdges pointer or its target vertex's enteringEdges pointer could point to it. In these situations, that vertex's pointer should be moved to the next vertex in the list.

4 PSP

Don't forget your PSP work!

5 Handing in your code

To handin your MP9 code, use the command

```
handin cs225 mp9 graph.C graphalgs.C
```