Week 05 Lab Exercise

Graph ADT Re-implementation

Objectives

- to implement a Graph ADT using adjacency lists
- to get some more experience with manipulating ADTs
- to carry out your own testing

Admin

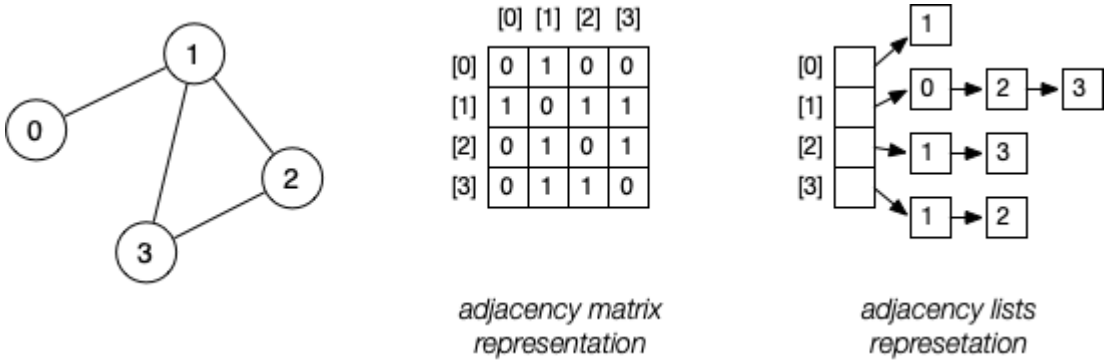
Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week 5 or Week 7 Lab
Submit	`give cs2521 lab05 Graph.c` or via WebCMS
Deadline	submit by **11:59pm Sunday 12 July**

The Problem

In the Week 04 online sessions, we looked at a simple interactive system for manipulating graphs. This program (`glab`) loaded up a graph either with random edges, or from a set of edges given in a file. It then provided commands to perform operations like insert a new edge, remove an edge, find a path from vertex A to vertex B, etc. We have supplied you with a working version of `glab`. "Ok, problem solved", you say, "What's left for me to do?".

The boss of the company who built `glab` has noticed a problem: it does not scale well to graphs with a large number of vertices and relatively few edges (sparse graphs). Because it uses an adjacency matrix representation, if you have a graph with 1000 vertices, the matrix will have 10^6 cells, most of which contain 0 in a sparse graph. The boss wants *you* to change the graph ADT so that it stores graphs using an *adjacency list* representation instead of an adjacency matrix.

The diagram below shows the two different representations for a small graph:



The lecture slides have more information about how graphs are represented, if you need it.

Getting Started

Set up a directory for this lab, change into that directory, and run the following command:

```
$ unzip /web/cs2521/20T2/labs/week05/lab.zip
```

If you're working at home, download `lab.zip`, unzip it, and then work on your local machine.

If you've done the above correctly, you should now find the following files in the directory:

`Makefile`	a set of dependencies used to control compilation
`glab.c`	a command-line interface to the Graph ADT
`Graph.c`	an implementation of a Graph ADT
`Graph.h`	the interface for the Graph ADT
`List.c`	an implementation of a linked-list ADT
`List.h`	the interface for the linked-list ADT
`Stack.c`	an implementation of a stack ADT
`Stack.h`	the interface for the stack ADT
`Queue.c`	an implementation of a queue ADT
`Queue.h`	the interface for the queue ADT
`PQueue.c`	an implementation of a priority queue ADT
`PQueue.h`	the interface for the priority queue ADT
`Item.h`	type of items stored in above ADTs
`graphs`	some example graphs

This may look like a lot of files, but you only need to look at a couple of them to complete this exercise. The files `Graph.c`, `Graph.h` and `List.h` are relevant. And you only need to modify one: `Graph.c`.

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce the `glab` executable.

The `glab` program reads commands from the terminal, and carries them out. After each command, it displays the new version of the graph.

Here's a sample session with `glab` that shows the kinds of things you can do:

```
$ ./glab 3
// create a random graph with 3 vertices
// it just happens to have no edges created
```

```
Graph has V=3 and E=0
V    Connected to
--   ------------
0
1
2

> i 0 1
// add the edge (0,1)

Graph has V=3 and E=1
V    Connected to
--   ------------
0    1
1    0
2

> i 1 2
// add the edge (1,2)

Graph has V=3 and E=2
V    Connected to
--   ------------
0    1
1    0 2
2    1

> i 2 0
// add the edge (0,2)

Graph has V=3 and E=3
V    Connected to
--   ------------
0    1 2
1    0 2
2    0 1

> ?
Commands:
 (i)nsert edge ... i v w
 (r)emove edge ... r v w
 (d)epth first ... d v
 (b)readth first ... d v
 (p)ath check (bfs) ... p v
 (P)ath check (dfs) ... p v
 (f)ind short path ... f v w
 (c)ycle check
 (C)onnected components
 (q)uit
```

```
Graph has V=3 and E=3
V    Connected to
--   ------------
0    1 2
1    0 2
2    0 1

> f 0 2
// find a path from 0 to 2
// note: the path is displaysed in reverse order
x=0, q=1
Path: 2<-0

Graph has V=3 and E=3
V    Connected to
--   ------------
0    1 2
1    0 2
2    0 1

> q

$ ./glab graphs/complete
// create a new graph ...
// using the edge info in graphs/complete
Graph has V=6 and E=15
V    Connected to
--   ------------
0    1 2 3 4 5
1    0 2 3 4 5
2    0 1 3 4 5
3    0 1 2 4 5
4    0 1 2 3 5
5    0 1 2 3 4

> f 0 5
// find a path from 0 to 5
x=0, q=1>2>3>4
Path: 5<-0

Graph has V=6 and E=15
V    Connected to
--   ------------
0    1 2 3 4 5
1    0 2 3 4 5
2    0 1 3 4 5
3    0 1 2 4 5
4    0 1 2 3 5
```

```
5    0 1 2 3 4

> r 0 5
// remove the edge (0,5)
Usage: p v1 v2

Graph has V=6 and E=14
V    Connected to
--   ------------
0    1 2 3 4
1    0 2 3 4 5
2    0 1 3 4 5
3    0 1 2 4 5
4    0 1 2 3 5
5    1 2 3 4

> f 0 5
// find a path from 0 to 5 (without direct link)
x=0, q=1>2>3>4
x=1, q=2>3>4>2>3>4
Path: 5<-1<-0

Graph has V=6 and E=14
V    Connected to
--   ------------
0    1 2 3 4
1    0 2 3 4 5
2    0 1 3 4 5
3    0 1 2 4 5
4    0 1 2 3 5
5    1 2 3 4

> q
```

A very common error that I make using `glab`, is to try to use the `d` command to remove an edge. If you do, you'll get an `assert()` error. To remove an edge, use the `r` command.

Your Task

Your task is to modify `Graph.c` so that it represents graphs using adjacency lists. You may use any of the other supplied ADTs in doing this; the `List` ADT will clearly be useful. You should not need to change any files apart from `Graph.c`.

Because the Graph ADT has been reasonably well-designed, you don't need to make many changes. You do need to change the `struct GraphRep` definition and modify to the following functions `newGraph()`, `dropGraph()`, `insertE()`, `removeE()`, and `adjacent()`. This may sound like a lot of changes, but it amounts to a total of only 10-20 lines of code.

Testing

How should you test your work? Compile `glab`, and then run some sessions with graphs of different sizes and different structures. We are not supplying a testing framework in this lab because we want you to think about what tests are required. It is clearly useful to look at inserting valid edges, but you should also look at what happens when you insert an edge that already exists or try to remove an edge that is not in the graph. You can try using invalid vertex numbers, but these will typically be detected by an `assert()` before they reach your code. It is also useful to check that algorithms like finding-a-path still work.

The critical point is to be able to create new graphs, insert edges, and remove edges. Checking creation means starting `glab` and seeing whether it produces a valid graph; to ensure that this is working, it is best to load graphs from files where you know what edges should be in the graph. You can check insertion and removal by running `i` and `r` commands and making sure that the updated graph is consistent with what you just did. If you keep a copy of the supplied `glab`, you can use that as a reference for what is supposed to happen.

You don't need to submit evidence of testing, but you *must* be able to explain what testing you did when you show your work to your tutor. Keeping a list of test cases in a file in the lab directory would help with this.

Submission

You need to submit one file: `Graph.c`. Submit it from the command line via `give` or via Webcms3. You should try to finish it in Week 05, but, if you can't, you can submit it at the end of the Flexibility Week. You should then show your tutor in Week 07, who will give you feedback on your coding style and your testing, and award a mark.

Have fun, *jas*