SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2017
Assignment 1: Road Network Traversal

Due: 12 noon, Monday 3 April

Overview

Navigating our world by car involves traversing a vast network of destinations connected by roads. What is the most efficient way to drive from point A to point B, taking distance, speed limits, and road and traffic conditions into account? Are there alternative routes available? These questions plague all motorists; from tourists and holiday-makers to commuters and couriers.

These road networks can be elegantly represented as a graph: with a vertex for each destination and distance-weighted edges for the roads connecting them. This abstraction allows us to unleash a large body of formal study regarding graph algorithms onto real world problems of navigation.

The state of Victoria and its sprawling network of major towns provides a perfect playground for us to explore these problems. In this project, you will implement several algorithms to assist with navigating the state of Victoria by road.

The project consists of a 5-part coding component and a 2-part written report. Each task is described in detail in the sections below.

Coding Tasks

Before you attempt the 5 coding parts of this assignment you will need to download the following files from the LMS.

`Makefile`	Edit to include your student number.
`vicroads.txt`	Do not change.
`main.c`	Do not change.
`graph.h, graph.c`	Do not change.
`traverse.h`	Do not change.
`traverse.c`	Complete with your own implementation.
`list.h, list.c`	Empty; replace with your own implementation.
`queue.h, queue.c`	Empty; replace with your own implementation.
`stack.h, stack.c`	Empty; replace with your own implementation.

At the top of any files that you edit, you should add your name and student number in a code comment (see `main.c` for an example).

At this point, you should be able to compile the supplied code (above) by running `make`. Once your project is completed, you will be able to execute it by running a command of the form

```
a1 -p part -s start -d destination -f file
```

where

- `part` is a number between 1 and 5 representing the part of the assignment to execute,
- `start` is a starting town index (required for all parts),
- `destination` is a destination town index (required only for parts 3, 4, and 5), and
- `file` is the name of a text file in the current directory describing a road network in a format understood by `main.c`.

For example, the command `a1 -p 1 -s 0 -f vicroads.txt` will run your solution to part 1 on the road network described in `vicroads.txt`, starting at town 0.

Data Structures

Your solution will need to work extensively with the data structures defined in `graph.h`. A `Graph` is used to represent a road network, with a `Vertex` array representing its towns. Towns are commonly referred to by their index in this array. Each `Vertex` is labelled with its town name, and holds the first `Edge` in an adjacency list. Each `Edge` represents a road weighted with its distance in kilometers, and links to the next `Edge` in the list. This list can easily be processed using a `while` or `for` loop.

You will find many places where using an abstract collection such as a stack or queue will simplify your implementation. Targets for list, queue and stack modules have been added to the project makefile, and blank source files have been provided. It's recommended that you develop these modules before or alongside the coding tasks. Your work from the week 2 and 3 labs will be helpful here. Lab solutions will be released in the weeks after each lab, and may be used in your project with proper attribution.

You may also create any additional modules necessary to support your solution. If you do add new modules, it is your responsibility to correctly extend your makefile—you must ensure that your solution can be compiled after submission simply by running `make`.

Part 1: Depth-First Traversal (1 mark)

Implement the function `print_dfs()` defined in `traverse.c`. This function takes as input a connected graph representing a road network, and the index of a starting town. The function should print the contents of the network as seen by a depth-first exploration starting from the given town. To ensure consistent output, process neighbouring towns in order of their appearance in the vertex edge list.

The function should print to `stdout` one line with the name of each town, in the order they are visited.

Part 2: Breadth-First Traversal (1 mark)

Implement the function `print_bfs()` defined in `traverse.c`. This function takes as input a connected graph representing a road network, and the index of a starting town. This function should print the contents of the network as seen by a breadth-first exploration starting from the given town. To ensure consistent output, process neighbouring towns in order of their appearance in the vertex edge list.

The function should print to `stdout` one line with the name of each town, in the order they are visited.

Part 3: Finding a Detailed Path (1 mark)

Implement the function `detailed_path()` defined in `traverse.c`. This function takes as input a connected graph representing a road network, the index of a starting town, and the index of a destination town. The function should find any simple path from the starting town to the destination town (a simple path does not contain any repeated vertices). The function should print this path, along with the cumulative distance of each town along the path (measured from the starting town).

The function should print to `stdout` one line for each town on the path, from start to destination. Each line should contain the town name, a space, and then the cumulative distance along the path in parentheses.

Part 4: Finding All Paths (2 marks)

Implement the function `all_paths()` defined in `traverse.c`. This function takes as input a connected graph representing a road network, the index of a starting town, and the index of a destination town. The function should find and print all simple paths between the starting town and the destination town (a simple path does not contain any repeated vertices).

Each path should be printed to `stdout` on a single line, as a list of town names each separated by a comma and a space. There should be no comma or space after the final town in each path.

Part 5: Finding the Shortest Path (2 marks)

Implement the function `shortest_path()` defined in `traverse.c`. This function takes as input a connected graph representing a road network, the index of a starting town, and the index of a destination town. The function should find and print the path from the starting town to the destination town with the shortest total distance. If there are multiple such paths, any one of them may be printed.

This path should be printed to `stdout` on a single line, as a list of town names separated by a comma and a space. After the final town should be a space and then the total path distance in parentheses.

Written Tasks

The final two parts of the project require a short, written report addressing the topics described below.

Your report must be no more than a single page in length, and must contain your name and student number at the top. The file should be named `report.pdf` (you may use any document editor to create your report, but you must export the document in `.pdf` format for submission).

Part 6: Graph Representation (1 mark)

The graph module provided by `graph.c` and `graph.h` stores graphs using an adjacency list representation. Describe any changes you would have to make to your implementations of depth-first traversal and breadth-first traversal (parts 1 and 2) had the graph module provided you with an adjacency *matrix* representation instead. You should describe these changes in English (rather than C code), but you may use short snippets of C code or pseudocode to assist your explanation.

Part 7: Design of Algorithms (2 marks)

Consider your algorithms for solving parts 3, 4, and 5. Briefly describe how these algorithms work. Your description should focus on how your algorithms traverse the graph to find the right path/paths, and on your use of data structures to find, record, and print these paths, rather than explaining each line of code in detail.

Bonus Marks

Two bonus marks are available for students seeking an additional challenge. These marks will be awarded to students who successfully complete the following challenges.

- **Removing Recursion (1 mark)** Implement part 1 (Depth-First Traversal) using an explicit stack instead of recursion. These changes should also be reflected in your answer to part 6.
- **Complexity Analysis (1 mark)** Add to your response to part 7 (Design of Algorithms) an analysis of the time complexity of each of your algorithms for parts 3, 4 and 5. Your analyses may carry over onto a second page in your report.

Sample Output

We provide some basic samples of correct output for each part of the assignment. Download the `samples` folder from the LMS. The files contain correct output from a selection of commands, as described in the table below.

These examples are intended to help you confirm that your output follows the formatting instructions. Note that there may be multiple correct outputs for some inputs. Note also that these samples represent only a subset of the inputs your solution will be tested against after submission: matching output for these inputs doesn't guarantee a correct solution.

Filename	Command
`vic-p1-s12.txt`	`a1 -p 1 -s 12 -f vicroads.txt`
`vic-p2-s12.txt`	`a1 -p 2 -s 12 -f vicroads.txt`
`vic-p3-s12-d4.txt`	`a1 -p 3 -s 12 -d 4 -f vicroads.txt`
`vic-p4-s12-d4.txt`	`a1 -p 4 -s 12 -d 4 -f vicroads.txt`
`vic-p5-s12-d4.txt`	`a1 -p 5 -s 12 -d 4 -f vicroads.txt`

Submission

Via the LMS, submit a single archive file (e.g. `.zip` or `.tar.gz`) containing **all files required to compile your solution (including the `Makefile`), plus your report (in `.pdf` format).** The archive should unpack into a folder, where the folder is named using your student number. Your submission should compile on the School of Engineering student machines (a.k.a. dimefox) without any errors, simply by running `make` inside this folder.

To make submission easier, we've added a `submission` target to the makefile. You need to locate the line `STUDENTNUM = <STUDENT NUMBER>` and include your student number. Then, running `make submission` will create the required archive file. If any files are missing, you will see an error.

Submissions will close automatically at the deadline. As per the Subject Guide, the late penalty is 20% of the available marks for this project for each day (or part thereof) overdue. Note that network and machine problems right before the deadline is not a sufficient excuse for a late or missing submission. Please see the Subject Guide for more information on late submissions and applying for an extension.

Marking

The five coding parts of the assignment will be marked as follows. You will score full marks for a part if your solution produces correct output for all inputs tested. You will lose partial marks for minor discrepancies in output formatting, or minor mistakes in your solution. You will score no marks for a part if your solution crashes on certain inputs, produces wrong answers, or causes compile errors on the School of Engineering student machines (a.k.a. dimefox).

Additional marks will be deducted if your code is difficult to interpret (due to missing or unhelpful comments, obscure variable names, poor functional decomposition, etc.), or if your solution has memory leaks.

The two written parts of your assignment will be marked as follows. You will score full marks for a clear and accurate solution that fully addresses the problem. You will lose partial marks for minor inaccuracies, or misuse of terminology or notation. You will score no marks for a generic discussion that fails to address the problem in the context of *your* algorithmic approach to parts 1-5.

Additional marks will be deducted if your report is too long, is not in `.pdf` format, or does not contain your name and student number.

Academic honesty

All work is to be done on an individual basis. Any code sourced from third parties must be attributed. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide for more information.