# ELE00028I: Algorithms and Numerical Methods Coursework Assessment 2022/2023

Y3903788

April 2023

**Abstract**

This report aims to describe the rationale behind the design of a graph abstract data type and the implementation of an algorithm to find the shortest path between two points on this graph, even on a data set that contains negative cycles.

## Contents

# 1 Design

## 1.1 Graph ADT

The graph implementation consists of three abstract data types. These are:

- **Graph**, which contains:
  - int **nodeCount** - The number of nodes in the graph.
  - int **edgeCount** - The number of edges in the graph.
  - Node** **nodes** - A list of pointers to **Node** objects. This contains every node/city in the graph.
- **Node**, which contains:
  - int **id** - The id corresponding to the index of the city in the edge list and hash table.
  - int **edges** - The total number of edges connected to the node. Defined by the size of edgeList.
  - Edge** **edgeList** - A list of pointers to **Edge** objects.
- **Edge**, which contains:
  - int **destination** - The id of the edge's destination node.
  - int **weight** - The weight of the edge.

The main function used to interact with the graph are:

- **createGraph** - The graph constructor. Returns a pointer to a **Graph** object.
- **destroyGraph** - The graph destructor. Frees all memory allocated by the graph.
- **addNode** - Allocates memory for a new node, then initialises all of its values. Subsequently adds a pointer to this node to the graph and increments the node count.
- **addEdge** - Takes the ids of two nodes and the weight of the connection between them. Reallocates additional memory for the edge list, then adds an edge object to both nodes. The object needs to be added to both nodes since the graph is bidirectional. Increments edge count on the graph and node.
- **getConnectionWeight** - Takes the ids of two nodes and returns the weight of the connection between them. If the nodes are not connected, the function will return **INT_MAX**.

## 1.2 Hash Table

Despite being somewhat beyond the scope of this assignment, implementing a hash table to store the city names and corresponding ids was a nice addition to the graph implementation, although at the scale of the graph set in the assignment (only 24 nodes), the use of a hash table provides no meaningful benefit over a simple lookup table. It was, however, interesting to learn about and implement.

**hashtable.c** allows for the creation of a hash table of a specified size, then populating it with elements by passing strings to the **addEntry** function. The table uses the djb2 hashing algotithm, which is a very simple hashing algorithm that has been found to generate relatively low collisions. Collisions are not a major concern with the graph provided but could slow down use of the table for much larger graphs.

### 1.3 Path finding algorithm

The graph provided for this assignment poses a major issue when it comes to implementing a shortest path algorithm. The presence of not only negative edges, but negative cycles would usually make finding the shortest path impossible. The assignment avoids this issue by specifying that each city may only be visited once, preventing negative cycles from occurring.

Since negative cycles exist within the graph and each node can only be visited once, the only option to find the shortest path between two nodes is to recursively traverse the graph until every path between the two nodes is found, compared and the shortest path recorded. This is accomplished through the **recursiveDFS** function, which takes the following arguments:

- **Graph*** - A pointer to the graph
- **int src** - The node to begin the traversal from.
- **int dst** - The final destination of the traversal.
- **int* visited** - A list of size nodeCount. Once a node has been visited, **visited[nodeIndex]** is set to 1 (0 by default).
- **int* currentPath** - A list of predecessors for each node, which can be traced back to form the path travelled.
- **int* shortestPath** - The shortest path encountered so far. Copied from **currentPath**.
- **int* shortestWeight** - The weight of the shortest path. This is compared against every time **dst** is reached, and changed along with **shortestPath** if a shorter path is discovered.

The function travels down each edge in each edgelist for each node, until it has travelled down every possible path in the graph. It does this by iterating through each edgelist and calling itself, passing on all of its initial arguments apart from **src**. The use of this depth-first search algorithm gives the program a time complexity of $\mathcal{O}(|V|!)$, which for this relatively small graph is acceptable, but implementing this solution with a much larger graph would take a significant amount of time, especially compared to conventional shortest path algorithms such as Djikstra or Bellman-Ford, which have a time complexity of $\mathcal{O}(VE)$ and $\mathcal{O}(|E| + |V| \log |V|)$ respectively. These time complexities scale much better than the factorial time implemented here.

## 2 Testing

### 2.1 Memory Usage

Listing 1 shows the Valgrind trace of the program. This trace proves that the program frees all memory it allocates during runtime, hence no memory leaks are possible. Under 21KB of memory is allocated throughout the runtime of the program, with 297 allocs and frees used. This is relatively good considering the program traverses and measures $> 10,000$ individual paths for each city pair.

### 2.2 Speed

When run with the **time** UNIX command, the time taken to execute the program and find the shortest path for all 3 city pairs can be measured to be 63 milliseconds.

```
==223581== Memcheck, a memory error detector
==223581== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==223581== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==223581== Command: ./main energy-v23-1.txt citypairs.txt
==223581==
Path from Manchester to Perth : Total W : 65
Manchester->Sheffield (Weight: -61)
Sheffield->Nottingham (Weight: 61)
Nottingham->Birmingham (Weight: -77)
Birmingham->Liverpool (Weight: 126)
Liverpool->Carlisle (Weight: -30)
Carlisle->Moffat (Weight: 65)
Moffat->Glasgow (Weight: -28)
Glasgow->Edinburgh (Weight: -74)
Edinburgh->Perth (Weight: 83)

Path from Liverpool to Whitby : Total W : -135
Liverpool->Carlisle (Weight: -30)
Carlisle->Manchester (Weight: 20)
Manchester->Sheffield (Weight: -61)
Sheffield->Leeds (Weight: 53)
Leeds->York (Weight: -39)
York->Whitby (Weight: -78)

Path from Lincoln to Birmingham : Total W : 58
Lincoln->Sheffield (Weight: 74)
Sheffield->Nottingham (Weight: 61)
Nottingham->Birmingham (Weight: -77)

==223581==
==223581== HEAP SUMMARY:
==223581==     in use at exit: 0 bytes in 0 blocks
==223581==   total heap usage: 297 allocs, 297 frees, 20,571 bytes allocated
==223581==
==223581== All heap blocks were freed -- no leaks are possible
==223581==
==223581== For lists of detected and suppressed errors, rerun with: -s
==223581== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Listing 1: Valgrind Output

## 2.3 Invalid Input

The application takes a limited amount of user input, in the form of two command line arguments, the first being the energy file, the second being the city pairs file. The **io.c** file contains the functions that handle these input files.

If **io.c** encounters any issues (invalid file format, invalid file path), it should print a message to stderr, and return **EXIT_FAILURE** from the main function..

- Invalid File Path:

```
Failed to open file.
Failed to open file.
Failed to count cities.
Failed to load energy.
```

- Invalid energy file:

```
Energy file not formatted correctly.
      Failed to load energy.
```

- Invalid city pairs file:

```
  Invalid citypairs file.
Failed to load city pairs.
```