

Algorithms & Numerical Methods Assignment

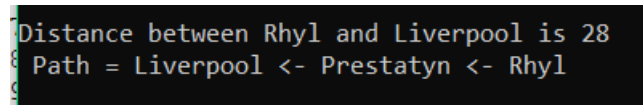
Introduction

shortestPath.cbp is a program that reads in a text file (ukcities.txt) that contains information which represents roads with distances that connect cities. The information from the file is then stored in memory, and then a second file(citypairs.txt) is read in, this time containing on each line two cities. The program will then calculate the shortest distance between the two cities on each line of citypairs.txt, then write the path and the total distance to a third text file, called paths.txt.

Design Choices and Justifications

I have designed this program to be as flexible as possible. As well as the user being able to change their intended paths in the citypairs.txt file, they can also add their own roads into the ukcities.txt file. For example, you could add a road to a new city by simply adding at the bottom of ukcities.txt

"[new city name] [new city name/already existing city] [distance]". For arguments sake, lets say that we add the roads "Rhyl Prestatyn 3", and "Prestatyn Liverpool 25". In the citypairs file, we could also add the line "Rhyl Liverpool" to find the distance from Rhyl to Liverpool. After doing this and running the program, the path from Rhyl to Liverpool will then obviously be printed as: "Path = Liverpool <- Prestatyn <- Rhyl" with the total distance being $3+25 = 28$ km.



```
Distance between Rhyl and Liverpool is 28
Path = Liverpool <- Prestatyn <- Rhyl
```

This is a screenshot of what is written to the "routes.txt" file after adding the lines as mentioned above.

Abstract Data Types

From information collected from Lecture 1¹, we have learned that abstract data types (ADT's) are primarily used in programming languages to "manage the complexity"² of solving algorithmic problems. They are also useful to hide the true complexity of the code. This is called the process of abstraction, in which the details of the implementation are hidden in a separate .c file. The data structures should not be directly accessed either, therefore a .h file is used to "describe the interface of the ADT"³.

For my code, I have created two ADT's, called "intertown_distance", and "citypairs". intertown_distance is an abstract data type that is used to store the information of the roads between cities from the ukcities.txt text file. The citypairs structure is used to store the start and end cities from the citypairs.txt file. The structures are represented below as:

```
struct intertown_distance{
    char start[256];
    char end[256];
    int distance;
};

struct city_pairs{
    char start[256];
    char end[256];
};
```

It is possible for the start and end fields of both structs to be allocated dynamically. This would make the program more efficient and also use less memory. However, I chose to define the arrays statically with a value of 256. I have decided not to dynamically allocate the char arrays as there is an increase in complexity in accessing these arrays while inside the structs, and I couldn't figure out a method of changing and reading the values of the dynamic arrays. It is therefore a safe assumption to make that a city's name will never be more than 256 characters⁴, using such a high value also provides a safety buffer to ensure that there is plenty of room if a new city with a particularly long name is added.

¹ https://www.elec.york.ac.uk/internal_web/meng/yr2/modules/Algorithms_and_Numerical_Methods/DSA/Lectures/Lecture1-AbstractDataTypes.pdf

² <https://www.quora.com/What-are-abstract-data-types>

³ Slides 26 and 27 of Lecture 1, linked in reference 1

⁴ https://www.elec.york.ac.uk/internal_web/meng/yr2/modules/Algorithms_and_Numerical_Methods/DSA/Lectures/Lecture9-Heaps.pptx - Slide 92

In the main function of the program, I have created two arrays of structures, one for `intertown_distance` and one for `citypairs`. They are defined as shown:

```
struct intertown_distance* roads[1024];
struct citypairs* citypairs[100];
```

I have defined the length of the roads array to be 1024. I have done this to allow future expansion of the program by allowing the addition of up to 1024 roads, which is a reasonable buffer. I decided to choose a smaller value for the citypairs array, because there wouldn't be much purpose in finding the shortest path between 100 or more different places at one time.

I have chosen to place the functions to interact with these data structures inside the implementation .c file, to hide the complexity from the user. This is done to disallow direct access to the data structures.

The Most Efficient Solution to the Problem

We were introduced to graph ADT's in Lecture 11⁵, which are a way to represent "pairs of objects connected by links"⁶. The basic functions that can be applied to a graph abstract data type would be for example:

- Add Vertex
- Add Weighted Edge

Therefore, the most efficient solution for the problem described in the specification would be to use a Graph abstract data type that stores the different cities as vertices, and the roads as weighted edges. The next step would be to then create an adjacency matrix/adjacency list from the graph structures, which can then be implemented into dijkstra's algorithm.

For my solution, I have opted to use a slightly less efficient method of implementing the graph functionality, in which I have created an adjacency matrix by using methods learned from the adjacency Matrix section on www.log2base2.com⁷. Then, using functions with the same functionality as the Graph ADT operations but with different implementations, I can add vertices and edges to the adjacency matrix.

Screenshots of my implementation of the graphs:

```
int graph[MAX][MAX]; - Creating the adjacency matrix
```

```
void initialiseGraph(int graph[MAX][MAX]){
    int i,j;
    for(i = 0; i < MAX; i++)
        for(j = 0; j < MAX; j++)
            graph[i][j] = 0;
}
```

- This function fills the graph with zeros, where the zeros mean that there is no link between two nodes, and MAX is a constant defined as 100

⁵ https://www.elec.york.ac.uk/internal_web/meng/yr2/modules/Algorithms_and_Numerical_Methods/DSA/Lectures/Lecture11-2019.SpanningTreesBEFOREDesignWorkshop.pptx

⁶ https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm

⁷ <https://www.log2base2.com/data-structures/graph/adjacency-matrix-representation-of-graph.html>

```
void addEdge(int graph[MAX][MAX], int source, int destination, int distance){
    graph[source][destination] = distance;
    graph[destination][source] = distance;
}
```

The addEdge() function above adds an edge to the adjacency matrix, taking the arguments of the starting city, the end city, and then the distance between the cities. The implementation for the graph adjacency matrix was adapted from the example code on log2base2.com⁸. After several attempts, I could not work out the correct way to implement the graph data structure, which encouraged me to simply create the adjacency matrix and load the weighted edges directly.

I decided to implement an adjacency matrix rather than an adjacency list because an adjacency list is a “more space-efficient way to implement a sparsely connected graph”⁹ – interactivepython.org. Also, as stated by a document published by Cornell University, “adjacency matrixes are better for dense graphs”¹⁰. This means that an adjacency list is more efficient for a sparsely populated graph, with few edges. However, in our case, there are 40 edges and 20 different cities. This means that the graph of cities and roads will be somewhat dense, which is the justification for my choice of using an adjacency matrix.

Dijkstra’s Algorithm

Brilliant.com¹¹ states that Dijkstra’s algorithm “creates a tree of shortest paths from the source node to all the other points in the graph”. This means that Dijkstra’s algorithm will find the shortest path from any given node to all other nodes. There are various implementations of Dijkstra’s algorithm, and the most efficient method that contains a priority queue. As stated by Tutorialspoint¹², the priority queue is a method that “queue items are ordered by key”, and the highest value of the key is at the front or the end of the queue. This is especially useful for Dijkstra’s algorithm, as the time complexity of Dijkstra’s algorithm that implements a priority queue is more efficient than the original Dijkstra’s without a priority queue¹³. GeeksForGeeks.com states that the “time complexity for the adjacency matrix is $O(V^2)$ ” and that the “adjacency list priority queue is $O(V+E)$ ”, where V is the number of vertices and E is the number of edges. The priority queue is used to search through each node’s connection, choosing to visit the nodes with the shortest distance and the highest priority first. Using this method, you can avoid having to brute force search through every single node until reaching a conclusion of the shortest path.

My implementation of Dijkstra’s algorithm is the original dijkstra’s algorithm that brute force searches through all elements in the graph. As explained above, this is not the most efficient method of Dijkstra’s algorithm, however the brute force search method was chosen as a result of my choice of graph structure. Creating a priority queue to be implemented in dijkstra’s algorithm requires an adjacency list, which my solution does not have.

⁸ <https://www.log2base2.com/data-structures/graph/adjacency-matrix-representation-of-graph.html>

⁹ <https://interactivepython.org/runestone/static/pythonds/Graphs/AnAdjacencyList.html>

¹⁰ <https://www.cs.cornell.edu/courses/cs211/2006fa/Lectures/L22-More%20Graphs/L22cs211fa06.pdf>

¹¹ <https://brilliant.org/wiki/dijkstras-short-path-finder/>

¹² https://www.tutorialspoint.com/data_structures_algorithms/priority_queue.htm

¹³ <https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/>

Heapsort

Heaps were first introduced in Lecture 9¹⁴, which states that a heap is a “binary tree if it is empty, or the root values are greater than or equal to the child values”.

Due to being unable to implement the correct priority queue minimum heap function, I have implemented a separate heapsort that sorts the roads data structure array in ascending order of distance. Below is a screenshot of the heapsorted ukcities:



```

Leicester Sheffield 100
Oxford Birmingham 103
Oxford Bristol 116
Blackpool Leeds 116
Sheffield Birmingham 122
Birmingham Manchester 129
Bristol Reading 130
Newcastle York 135
Birmingham Bristol 139
Carlisle Blackpool 140
Hull Nottingham 145
Edinburgh Carlisle 154
Liverpool Nottingham 161
Newcastle Edinburgh 177

```

As shown in the screenshot, the distances are sorted in ascending order.

Time complexity of the Solution

My solution implements a collection of different algorithms and recursion methods, each with their own time complexities. As explained by the user T.E.D on StackOverflow¹⁵, the Big O notation is about “which factor dominates as n approaches infinity.” Therefore, we can use the Big O Notation cheat sheet¹⁶ which states that the time complexities of:

Heap Sort = $O(N \log N)$, Dijkstras Algorithm = $O(V^2)$, array insertion/deletion = $O(n)$.

Where N is the number of operations. (In dijkstras case, N represents the number of vertices).

Therefore, as N tends to infinity, the dominant complexity would be $O(V^2)$. To prove this, you can simply choose a value for N and put it into the equations, for example $N = 1000$:

Heap sort = $O(1000 \log 1000) = 3000$, dijkstras = $O(1000^2) = 1000000$.

This clearly shows that $O(N^2)$ is the dominant time complexity of the program.

Limitations and Issues

A major limitation of my solution is that the execution time scales almost linearly with the number of paths to be calculated in the citypairs.txt file. For example, if there are three paths in citypairs.txt, the execution time will be about 0.1-0.2 seconds, and then if I put 6 paths in the file, the time to execute will be nearly doubled. An example of this is shown in the testing section (test 2). The slowdown is caused when writing strings to the paths file bit by bit, using the writeToPaths(char text[]) function, which takes the arguments of the text to be written to the paths file. Perhaps a better way of implementing this functionality would be to create a global array of char arrays, which stores the contents of the path data, which can then be passed to the writeToPaths() function only once.

¹⁴

https://www.elec.york.ac.uk/internal_web/meng/yr2/modules/Algorithms_and_Numerical_Methods/DSA/Lectures/Lecture9Heaps2018.pptx

¹⁵ <https://stackoverflow.com/questions/4934349/big-o-notation-for-multiple-functions>

¹⁶ <http://bigcheatsheet.com/>

Testing Methodology, with Results**Test 1 – Efficiency of the Algorithm**

Method – Running the program several times, recording the time taken to complete the program and then taking the average of the time taken, both with and without writing to the paths file.

Results:

Writing to File:

Attempt	Time(s)
1	0.117
2	0.103
3	0.100
4	0.095
5	0.095
Average	0.102

Without Writing to File:

Attempt	Time(s)
1	0.016
2	0.031
3	0.016
4	0.016
5	0.016
Average	0.019

Comments

The results clearly show that writing the path data to the paths file slows down the execution of the program significantly.

Test 2 – adding extra roads and paths to be calculated

Method – Adding additional roads to the end of ukcities.txt, and the path to test it on in citypairs.txt

```
Rhyl      Prestatyn      3
Prestatyn      Northampton 16
Denbigh Rhyl      15
Denbigh Prestatyn      12
Denbigh Leicester      125
```

- This has been appended to the ukcities file

```
Rhyl      Leicester
Denbigh Northampton
Leeds      Prestatyn
```

- This has been appended to the citypairs file

Results

```
Distance between Rhyl and Leicester is 80
Path = Leicester <- Northampton <- Prestatyn <- Rhyl

Distance between Denbigh and Northampton is 28
Path = Northampton <- Prestatyn <- Denbigh

Distance between Leeds and Prestatyn is 230
Path = Prestatyn <- Northampton <- Leicester <- Sheffield <- Leeds

Process returned 0 (0x0)   execution time : 0.193 s
Press any key to continue.
```

Above is the terminal output. This test shows that there is a severe limitation on my code, as the execution time has almost doubled from the first test. To test this further, I will add another 6 lines to my paths file, and I should expect that the execution time will be doubled again, to about 0.4s.

Results:

```

Distance between Sheffield and Doncaster is 29
Path = Doncaster <- Sheffield

Distance between Prestatyn and Oxford is 84
Path = Oxford <- Northampton <- Prestatyn

Distance between Carlisle and Northampton is 470
Path = Northampton <- Leicester <- Sheffield <- Leeds <- Blackpool <- Carlisle

Distance between Hull and Newcastle is 195
Path = Newcastle <- York <- Hull

Distance between Lincoln and York is 118
Path = York <- Doncaster <- Lincoln

Process returned 0 (0x0)   execution time : 0.578 s
Press any key to continue.

```

As shown in the screenshot, the execution time more than doubled this time. This highlights a major flaw in my solution to the problem.

Test 3 – Testing for memory leaks

Method – Try to print the adjacency matrix after attempting to free all memory associated with the structures.

In theory, the code below when placed below the attempted free memory function should print jargon, which would indicate that the memory has been correctly freed.

```

for(i = 0; i < getRoadsSize(roads); i++){
    printf("\n%s\t%s\t%d\n", getStartLocation(roads[i]), getEndLocation(roads[i]), getDistanceBetween(roads[i]));
}

```

Results:

```

Bristol Reading 130
Newcastle York 135
Birmingham Bristol 139
Carlisle Blackpool 140
Hull Nottingham 145
Edinburgh Carlisle 154
Liverpool Nottingham 161
Newcastle Edinburgh 177
Process returned 0 (0x0)   execution time : 0.157 s
Press any key to continue.

```

Comments -

As shown here, the memory hasn't been freed correctly as the details from city pairs are still present. This means that there is a memory leak that I am unsure of the reason. In my roads destructor function to free the allocated memory, nothing gets freed and the structs remain as they were.

```

void intertownDistanceDestructor(struct intertown_distance* intertown_distance){
    free(intertown_distance);
}

void destroyRoads(struct intertown_distance* roads[1024]){
    int i;
    for(i = 0; i < getRoadsSize(roads); i++){
        intertownDistanceDestructor(roads[i]);
    }
}

```

The destroyRoads function is called in the main, however when I try printing the contents it still works as if no memory has been freed. This is an issue I have not been able to fix, or understand why free() doesn't do anything in this particular case.

Test 4 – Testing the Heapsort

Method – printing the ukcities text file before and after applying a heapsort

Results:

Before Heapsort:

```

York Hull 60
Leeds Doncaster 47
Liverpool Nottingham 161
Manchester Sheffield 61
Reading Oxford 43
Oxford Birmingham 103
Birmingham Leicester 63
Liverpool Blackpool 79
Carlisle Newcastle 92
Nottingham Birmingham 77
Leeds York 39
Glasgow Edinburgh 74
Moffat Carlisle 65
Doncaster Hull 76

```

After Heapsort:

```

Sheffield Doncaster 29
Leeds York 39
Reading Oxford 43
Leeds Doncaster 47
Leeds Sheffield 53
York Doncaster 55
Liverpool Manchester 56
York Hull 60
Leicester Northampton 61
Nottingham Sheffield 61
Manchester Sheffield 61
Birmingham Leicester 63
Lincoln Doncaster 63
Manchester Leeds 64
Moffat Carlisle 65

```

Comments – As the screenshots show, the roads array has been sorted in ascending order of distances, as was intended.