ELE00028I Algorithms & Numerical Methods Assessment

Abstract

This report is written to include explanations of the design used to achieve the result showing the shortest path between pairs of specified locations.

Within this report there will be details of the choice of data structures, algorithms and interfaces used in the program. This will include the consideration of memory requirements as well as computational efficiency. Following this, the output from the program will be provided, and a detailed explanation of how the code was tested, and how features that the program has used to achieve the result.

At the end you will find an evaluation which includes a review of each feature in the specification stating and analysing the successful implementations as well as the limitations and unsolved problems (bugs). Furthermore, this will look at the overall quality and performance and finish with a conclusion stating some improvements for further adaptions.

Problem Analysis

Problem Description

The task set is to design and implement a program which constructs a graph of cities from any given text file containing a list of two cities connected together and an energy to travel between them.
This energy will connect the cities in the graph with a "weight" equal to the energy expended when travelling between those two city pairs. The program will then use a shortest path algorithm to provide the route of shortest energy expenditure between any given source and destination city pair.

Approach to Creation and Design Choices

Starting out, the main thought processes involved researching different shortest path algorithms and how they were used so that a couple which were suitable could be picked out and experimented with. This provided details of the efficiency and memory requirements of each algorithm as well as how well they could handle the specifics of our task (being able to handle negative weighted edges and potential negative cycles). Alongside this, the best data structures to use for each algorithm were also researched so that the importing of the text file could be implemented in the most efficient manner.

Three common algorithms that are used to calculate the shortest path were found being, Dijkstra's, Bellman Ford and Floyd Warshall. Dijkstra's was discarded due to not being able to handle negative edge weights – an important factor in this task. This left the Bellman Ford and Floyd Warshall algorithms to be considered.

Floyd Warshall

Considering the Floyd Warshall algorithm first, due to its simpler implementation, hence it is faster to start testing, the most common data structure to be used for this algorithm is an adjacency matrix. An adjacency matrix is a two-dimensional array which stores the graph concerned, at each position in the matrix the weight of each edge is stored and represents the energy between the ith and jth index in the matrix. (Where i and j are a source and destination city pair). Assuming the graph has n vertices, the time complexity to construct a matrix of this size is $O(n^2)$ since it requires 2 nested loops to iterate over each individual index. Since it's a square matrix the space complexity is also $O(n^2)$ as it it's dimensions are $n \times n$. Due to most graphs only having a few connections per city, this data structure is realistically less suitable since it therefore uses a lot more memory than required. However, operations such as adding edges and removing edges, as well as checking if there is an edge from i to j are very efficient, and every access operation into the matrix is always constant.
For an adjacency matrix to be used with the Floyd Warshall algorithm, it needs to be initialised with infinity for all unique city pairs (e.g. The edge from Liverpool to Manchester at

$$Matrix[Index\ of\ Liverpool][Index\ of\ Manchester] = Energy\ from\ Liverpool\ to\ Manchester$$

where the energy is initialised to infinity). For repeated cities, i.e. $Matrix[Liverpool][Liverpool]$, these must be set to zero, since there cannot be a path to itself. Having initialised the matrix, it can then be filled with the known city pairs, therefore creating the specific graph required.
The Floyd Warshall algorithm itself works by finding the shortest path between all pairs of vertices in the graph. Therefore, by the end of the algorithm, the matrix will be completely filled out with varying energy values to travel between each source and destination city pair. In order to achieve this, the implementation involves 3 nested "for" loops, and an intermediate vertex is used as a comparison route for the path being solved.
If k is let to be the intermediate vertex between a source i and destination j pair, then the algorithm iterates through each index for each value of k up to the number of vertices, and updates the distance from the source to the destination if the path going from i to k to j is shorter than the path from i to j, with the path through k.

$$If\ Matrix[i][k]\ +\ Matrix[k][j]\ <\ Matrix[i][j])$$

$$then\ Matrix[i][j]\ =\ Matrix[i][k]\ +\ Matrix[k][j];$$

Since there are three loops involved, with each loop having a constant complexity, the time complexity of the algorithm itself is $O(n^3)$, with the space complexity being $O(n^2)$ as mentioned previously.

As it currently is, the Floyd Warshall isn't much help for dealing with an undirected graph that contains negative edges. This is because any graph as such will mean that negative cycles exist, which are not desirable at all, since when a shortest path algorithm runs through the graph, if it finds a negative cycle it will continuously loop round the cycle hence infinitely reducing the paths energy until its negative infinity.
The solution to this uses a "path" array, which updates i to j with i to k when the previous condition above is met to reduce the path length. This is useful as it is subsequently used each time the algorithm goes through a loop to check whether a specific city on the route from source to destination has already occurred. This will ensure that the path from i to j does not contain any repeated city and so should avoid going over an undirected negative edge multiple times, hence eliminating any negative cycles.

<u>Bellman Ford</u>
An alternative approach to the Floyd Warshall algorithm is the Bellman Ford algorithm. This algorithm is more suitable for the overall problem being solved, since rather than calculating the path between all vertices like the Floyd algorithm, this algorithm calculates the shortest path from a given source vertex by the user. Therefore, paths which are not required to be known are not being calculated which uses unnecessary computational power and time.
The standard data structure used for the Bellman Ford algorithm is an adjacency list, which implements an array of linked lists, with each linked list in the array representing all the connections (edges) from one source city. An adjacency list is more complex to implement and understand than an adjacency matrix, however it does have a couple of advantages over the matrix.
An adjacency list only stores the values for each edge in the graph, it does not need to store each edge between all combinations of source to destination city pairs. So, for a sparse graph, this can save a lot of space. Secondly, the adjacency list is very fast to access all the vertices connected to a source vertex. However, finding the specific edge weight or energy between two specific vertices in the graph is much slower than an adjacency matrix, since all the connected nodes to a source must be iterated through to find the specified destination vertex.
From these details, the space complexity of the adjacency list is $O(|V| + |E|)$ for the average case (where "V" is the number of vertices and "E" is the number of edges in the graph), however the worst case can still be $O(V^2)$ since each vertex can be connected to every other vertex. Generally, this means the space complexity is better than an adjacency matrix, but depending on the type of graph it can be the same. As for the time complexity, for building the linked list it is $O(E)$, since you need to iterate through the nodes in the list to add a new edge onto the end of the list. A similar time complexity is also seen for checking the presence of an edge in an adjacency list, which is $O(V)$. Comparing this to the adjacency matrix, $O(V)$ is linear compared to the constant $O(1)$ time complexity to check for an edge in the matrix. Below is a table summarising the time complexities for various operations in both an adjacency matrix and an adjacency list.

Use Cases	Adjacency Matrix	Adjacency List
Initialising data structure	$O(V^2)$	$O(E)$
Adding a vertex	$O(V^2)$	$O(V)$
Adding an edge	$O(1)$	$O(E)$
Removing a vertex	$O(V^2)$	$O(V)$
Removing an edge	$O(1)$	$O(E)$
Checking the presence of an edge	$O(1)$	$O(V)$

Using the adjacency list with the Bellman Ford algorithm, the graph is first filled out and added into the list. Each source vertex is added into the array $graph{->}\ connectedCities[source]$ and the corresponding connected vertices are connected using nodes in a linked list: $graph{->}\ connectedCities[source].head{->}(destination\ and\ energy)$, where the array at index "source" points to the start of the linked list called "head". And the node "head" contains a variable to store the destination city, the energy between the city pair, as well as a pointer to point to the next node where the next destination city is stored. This is repeated for all the city pairs in the text file.

The algorithm can then be run. Firstly, two arrays are created an initialised, a "distance" array and a "predecessor" array. The size of these is equal to the number of vertices in the graph, and to initialise them the distance array is filled out with infinity at each index apart from the starting city (source) that you are finding the path from, which is set to 0. The predecessor array is also filled completely with 0. To find the shortest path, the algorithm iterates over each edge $|V| - 1$ times. Each time if the current source city plus the energy to that next node, is less than the current distance to the destination city, then the distance to the destination city is updated. The predecessor array also gets the source city at the destination city index, which builds up the path the algorithm travels through for the shortest path.

$$If\ distance[dest]\ >\ distance[src]\ +\ energy$$

$$then\ distance[dest]\ =\ distance[src]\ +\ energy;$$

Since the algorithm involves 2 loops, with both having a constant complexity, the average time complexity of the Bellman Ford algorithm is as follows: $O(VE)$. This can become a best time complexity of $O(E)$, depending on the number of vertices.

A minor modification can be made to the algorithm, to make sure that due to negative edges the distance array isn't updated when both the source and destination city are at infinity. The conditional check is as follows: $If\ distance[src]\ !=\ INFINITY$ then the distance array can be updated.

This is required, since the acutal infinity used is not infinte, it is just a very large number. So if the edge is negative, a large number minus an edge weight, would be less than the other large number. Whereas theoretically speaking, the values of infinity will be infinite, so subtracting a small number would still result in infinity.

Data Manipulation

Now that the algorithm being used is known, it's possible to decide upon a method to import all the data. Since both an adjacency matrix and adjacency list are being used, a suitable approach can be picked to import and store the data.

A linked list has been used for this since it can store each of the city pairs imported from the text file in a separate node per pair. The first modification which is made here is to convert each city to an integer so the data manipulation with the algorithms later is much easier. This can be achieved by using an associative array and storing each unique string city name at its own index in the array, and then that index refers to the integer equal to the city.

Using this feature, a couple of functions can be written to simplify the access of data within the linked list and array, so that code efficiency while operating on it during running the algorithms is improved.

Testing Methodology and Results

Implementing this program requires a lot of testing to get results which are as user friendly and accurate as possible. This section aims to provide an in-depth analysis of the testing methods used as well as results achieved throughout the testing process.

Having implemented the Floyd Warshall algorithm first, the initial testing carried out involved seeing the initial results with no further modifications. As expected for an undirected graph with negative edges, the results gave a very large negative number. To work out how the negative cycles were being caused. A smaller graph that focussed on four well connected cities was used instead of the full graph. This allowed testing to look directly at just a couple of routes rather than the hundreds in the larger graph.
Leeds, Doncaster, York, and Sheffield were used for this graph, and a new text file was created in the same format which listed the routes between those cities from the larger graph. A construction of the graph can be seen in Figure 1.

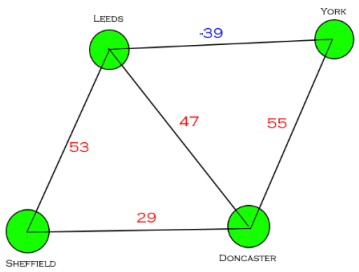

Figure 1 Diagram to visualise the smaller graph used for testing.

By running the Floyd Warshall algorithm for this graph, each process is simplified, and the specific routes being taken could be looked at. One of the problems found was that for the journey from Leeds to Doncaster, the shortest path it gave was -31 energy (Figure 2).

```
Shortest Path Graph

 0 -       Leeds: [    0,   -31,   -39,   -25]
 1 -   Doncaster: [  -31,     0,     8,    22]
 2 -       York: [  -39,     8,     0,    14]
 3 -   Sheffield: [  -25,    22,    14,     0]
```

Figure 2 Shortest path graph when the algorithm solves an undirected graph.

This is not correct as it can easily be seen from Figure 1 that the shortest path is Leeds to York to Doncaster equating to an energy expenditure of $-39 + 55 = 16$. But by doing some simple calculations and knowing the graph is undirected, it can be found that the algorithm was going from Leeds to York (-39), then it was checking what is the shortest path from York to Doncaster. The algorithm had already found that this shortest path was York to Leeds to Doncaster: $-39 + 47 = 8$, this meant that for the route from Leeds to Doncaster the algorithm was travelling from Leeds to York, then back to Leeds and then to Doncaster. Giving an energy expenditure of $-39 - 39 + 47 = -31$. As stated in the design brief, going back through a city i.e. travelling through a repeated city, is not allowed, therefore a solution needed to be found to fix this problem.

Since an the problem was due to the graph having undirected edges, which causes the negative cycle, an implementation could be tested where all the edges are made directed in one direction, and the shortest path calculated from that.
Testing this out on the test graph above gives an accurate result for Leeds to Doncaster as shown in Figure 3.

```
Shortest Path Graph

 0 -       Leeds: [     0,     16,    -39,     53]
 1 -   Doncaster: [ 99999,      0,  99999,  99999]
 2 -       York: [ 99999,     55,      0,  99999]
 3 -   Sheffield: [ 99999,     29,  99999,      0]
```

Figure 3 Shortest path from Leeds to Doncaster is 16 for a random direction directed graph.

Extract from the Output text file
Shortest path from source to destination city
Using floydwarshall (dir)
Leeds to York Energy = -39
Leeds to Doncaster Energy = 16
York to Doncaster Energy = 55

As can be seen from these results, most of them are accurate, however some of the distances are still at infinity, which is not correct, this is because some routes are not accessible. E.g. From Doncaster to anywhere else its infinity because all edges connected to Doncaster are directed towards it not away from it. Hence if this directed graph was expanded for the larger graph, the results would be very inaccurate and not representative of the shortest path.
To solve this, the edges need to be directed in the other direction too. So, the simplest implementation found

was to make the graph directed, run the algorithm, then make the graph reverse directed (i.e. directed in the opposite direction) and run the algorithm again. And then compare both the results and update the final graph and each path with the shortest path between the two resulting graphs.
Positive edges do not cause any negative cycles, so to save on the time complexity especially when comparing the directed and reverse directed results, rather than make every edge directed only the negative edges can be made directed so the negative cycles don't occur. After running this implementation, it can be seen in Figure 4, that the results are now much more accurate.

```
Finalised Shortest Path Graph

  0 -       Leeds: [     0,     16,    -39,     45]
  1 -   Doncaster: [    16,      0,      8,     29]
  2 -        York: [   -39,      8,      0,     14]
  3 -   Sheffield: [    45,     29,     14,      0]
```

Figure 4 Shortest paths are now shown accurately in both directions for each city.

Extract from the Output text file
Shortest path from source to destination city

Using floydwarshall (-vedir)

Leeds to York Energy = -39

Leeds to Doncaster Energy = 16

York to Doncaster Energy = 8

Testing for this stage has been made simpler by using command line arguments, which saves on modifying the code and risking adding in errors. These command line arguments are given as follows:

$$programName\ energyFilename\ citypairsFilename\ outputFilename graphDirection\ algorithmUsed\ verbose$$

In the testing above the arguments being used were:

$$Algorithms\_Assignment\ 4\_cities.txt\ 4\_citypairs.txt\ results.txt - vedir\ floydwarshall\ f$$

Most of these arguments are self-explanatory, however, "verbose" is an option which allows the program to be run with a either a verbose output or simplified output. Within the code there are various print statements to show the progress of each algorithm and the data structures, so by setting "verbose" to true, "t", when the program is run it will output all the stages where paths in the graph are reduced by the algorithm, as well as printing the data structures containing the input data as they are imported from the text file. If it is set to false, "f", then the program will only print out the key info to the terminal, such as the initial graph, and the final graph, as well as the index referring to each city.

As explained in the previous section, the Floyd Warshall algorithm is less efficient for this task due to it finding the shortest path between all city pairs, rather than just the specified pairs. To see if this can be improved the Bellman Ford algorithm has also been implemented to the specification described above. Since the testing for the Floyd Warshall with the directed graphs will be the same for the Bellman Ford, this was also implemented, and could as such be tested by specifying "bellmanford" for the algorithm argument rather than "floydwarshall". When the Bellman Ford is run for the "-vedir" argument, the results calculated are not as accurate, shown in Figure 5. This observation could be due to the different way in which the algorithm solves the shortest path. However, without sufficient time it seems very complex to be able to achieve a more accurate result. The differences are more visible once the full energy text file is used.

```
Distance array from source Leeds: 0 16 -39 53
Predecessor array: 0 2 0 0

Distance array from source Leeds: 0 99999 99999 99999
Predecessor array: 0 0 0 0

Distance array from source Leeds: 0 16 -39 53
Predecessor array: 0 2 0 0

Distance array from source Leeds: 0 99999 99999 99999
Predecessor array: 0 0 0 0

Distance array from source York: 99999 55 0 99999
Predecessor array: 0 2 0 0

Distance array from source York: -39 99999 0 99999
Predecessor array: 2 0 0 0
```

Figure 5 Shows the forward direction results then the reverse direction results for each of the three city pairs shown to the right

Extract from the Output text file
Shortest path from source to destination city

Using bellmanford (-vedir)

Leeds to York Energy = -39

Leeds to Doncaster Energy = 16

York to Doncaster Energy = 55

Now that both algorithms have been tested to give relatively accurate results, and in the case of the Floyd Warshall algorithm they appear to be 100% accurate for the smaller graph. The results from the full graph are now easier to analyse and should portray a similar accuracy as the smaller "testing" graph. Below is the final results for both the Floyd Warshall algorithm and the Bellman Ford algorithm, showing the terminal results as well as the text file outputs.

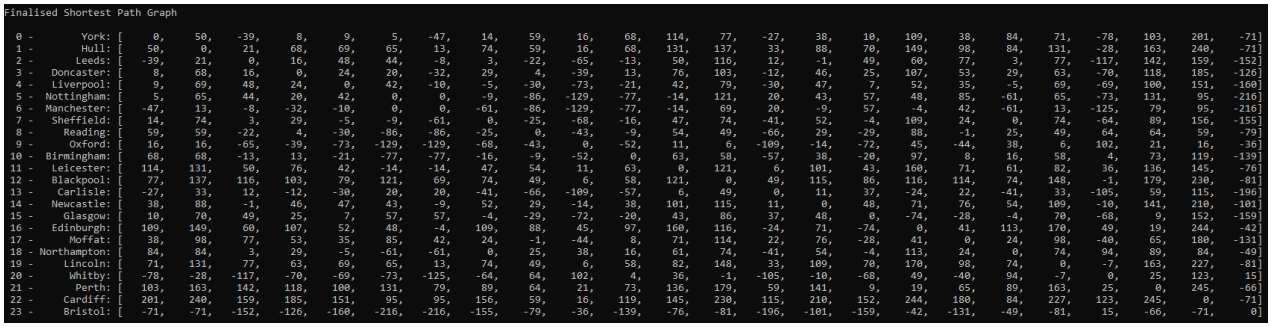

Figure 6 Floyd Warshall full graph output

Shortest path from source to destination city

Using floydwarshall (-vedir)

Manchester to Perth Energy = 79

Liverpool to Whitby Energy = -69

Lincoln to Birmingham Energy = 58

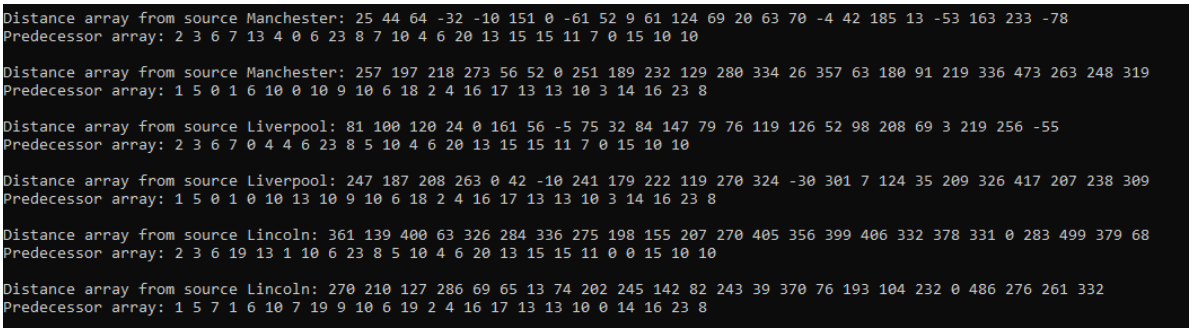

Figure 7 Bellman Ford full graph output

Shortest path from source to destination city

Using bellmanford (-vedir)

Manchester to Perth Energy = 163

Liverpool to Whitby Energy = 3

Lincoln to Birmingham Energy = 142

Evaluation

From looking at the overall graph, although it is very hard to manually calculate the shortest path, the results given by the Floyd algorithm seem reasonable and the ones which can be estimated match up with the results here. The Bellman Ford algorithm is not far off, though based on the testing above, it can be seen to not be 100% accurate, so these results although close to the Floyd Warshall are probably not the shortest path.
From looking at the execution time of the algorithms, (printing to the terminal needs to be taken into account, since that is very resource intensive), it can be seen that the Floyd algorithm runs fully in 0.576 seconds (prints out large matrices though), and the Bellman Ford algorithm runs fully in 0.119 seconds. Therefore, it can be said that both algorithms are very efficient and scale up to larger graphs well.

Each subsystem to the program was developed thoroughly and tested to a point which would meet a success for the program task. Within the program, where a small task or calculation was required, a function has been added to provide the feature. This has enabled the code to be well structured, and easy to understand how each section works. Additionally, the program will work for any set of strings and energy values, formatted in a tab delimited text file, that is given.

A further modification for a future improvement, which would greatly improve the memory complexity of the program, would be to modify the adjacency matrix to be dynamic, since it is currently static. This is a complex procedure and would ultimately require a lot more time to refactor the code, so it works just as efficiently.