**Algorithms Programming Assignment Report**

**Table of Contents:**

* **Abstract**
* **Program design**
  + **Description of problem**
  + **Data Structures**
  + **Shortest Path Algorithm**
* **Testing**

**Abstract**

In an era of rapid technological development, problems previously thought to be inconceivable due to the number of variables and complexity of interconnections can begin to be approached proactively and in great detail. For this assignment, a program was required which can calculate the “shortest” path from one UK city to another. With fuel-powered automobiles, the most energy efficient route (for the driver) is nearly always going to be the shortest in distance. However, with increasing focus on the development of electric vehicles, a world in which electric cars would be able to be charged through the roads themselves, can be envisioned. This alters the traditional approach to travel speeds and costs perhaps making a longer distance travelled the most appropriate route. This assignment requires a program to define a path that is as energy efficient (for the driver) as possible, with the consideration that some roads have been implemented with wireless charging technology.

**Program Design**

**Description of the problem**

The roads in between these cities have energy values which show how much energy is expended when driving between them. A text file which contains singular routes from one city to another along with an energy value is used to create a graph of cities. The cities make up the vertices (or nodes) of the graph, and the roads between them act as two-way edges. The graph is then used to calculate the route, which requires the minimum possible energy expended from one given city to another. Another file named “citypairs.txt” contains three pairs of cities which the program must produce a value for the shortest / most appropriate possible route for each.

**Data structures**

The program needed to be able to import data from the two text files provided in the brief, “energy.txt” and “citypairs.txt” so it could use the data they contained. The data stored in the files needed to be stored in memory, and if it can be stored so that the user doesn’t have top level access to change the values that would be ideal.

I opted to use double linked list structures to store the data. Two independent lists had to be created because the two files that were imported contained different collections of information. For the energy file, a linked list made up of nodes which stored energy, along with start and end cities, was required as the program needed to be able to interpret all 3 points of data consecutively in order to start to build a graph.

The “citypairs.txt” file consisted of source and destination cities which needed to be used by the program in order to calculate routes. This led to the implementation of a city linked list structure comprised of nodes which stored a pointer (along with pointers to adjacent nodes) to a charof a city name. Various instances of this structure feature in the program. Most notably perhaps is my “uniqueCities” linked list, which imports all the cities found in the energy file but makes allowance for duplicates, so a list of discrete cities is produced. This was made possible with the use of “strcmp()” which comes with the “<string.h>” library.

The two structure types mentioned needed to be allocated memory in theheap because they were being used globally. “Malloc()” was used for both (and their node structures) in constructor functions which dynamically calculated the amount of memory needed with the aid of the “sizeof()” function. Both structures were used in my “graphConstructor()” function which needed access to their node members. Attention must be paid to freeing up the allocated memory once it is no longer needed (after the main loop in “main()” ) to prevent a memory leak. The order in which a structure is freed is also important. To prevent any leaks, the memory associated must be freed in the reverse order it was reserved in (head node first, tail node last).

Once the data structures “energyList” and “cityList” had been established, along with all the necessary functions used to access them, the construction of the graph was next to be implemented.

Linear search functions

Linear search functions were needed to retrieve specific nodes in both linked lists as the index or the data would have been needed at some point, the time complexity of such functions is O(n).

Double link lists

When considering the data structures to store these sets of data, I had the choice of using array lists or double linked lists. Array lists have a superior time complexity of O(1) when accessing data members, but are equal to or inferior to the double linked list when considering search, insertion or deletion. Additionally, from a memory perspective, array lists have a set memory allocation, whereas double linked lists can dynamically add or release memory. This is important when the program needs to account for different sizes of text file to be imported. Another advantage of double linked lists is that the memory associated with them is allocated during runtime as opposed to array lists which have memory allocation executed when compiling. In summary my decision to use the heap dictated use of double linked lists to avoid use of stack memory.

The “cityList” structure was used to store the cities provided in the “citypairs.txt” in two lists, one for the source cities and another for the destination cities. Using the size value of these lists, the program can dynamically dictate the main for loop to repeat “size” times. This allows the program to operate with better modular functionality.

Graph

Two-way roads between cities are represented by two edges connecting them, one going one way, and one going the opposite way. Each line of data in the energy file provided all the information required for a set of two edges between two nodes. The “uniqueCity” list was used to populate the array of vertices (nodes). Each unique city represented one vertex. I wrote simple functions to return the integer size value for both “uniqueCity” and “energyList” lists, which would be doubled in the case of the edge array, and the face value for the vertices.

When it came to implement the structure for the graph, along with how many edges and vertices (nodes) it had, a pointer to an edge structure needed to be included too. A search of the unique city linked list was conducted so that the city member variables of the edge structure could be located, and their string recorded. The pointer to the string is then used to find the index. Once the number of edges was known, the memory allocated had to be doubled to allow for the two-way roads. Fortunately, the roads were ordered to have the same energy expended value in both directions, so it was a simple process doubling the number. However, when it came to assigning the values of the edge, a nested for loop was utilised to allow for the reversal of “start” and “dest” city values.

**Shortest Path Algorithm**

The problem this assignment presents requires the use of an algorithm, a logical process in which we can obtain the shortest path between two given cities. Perhaps the most well-known algorithm for determining the shortest path within two points of a weighted digraph structure is Dijkstra’s. However, as previously mentioned, some roads charge the car driving over them, which means that the edges associated with them have negative energy expended value. This renders Dijkstra’s invalid as it does not work with negative edge values. Looking further, the next reasonable option is the Bellman-Ford algorithm. It produces a shortest path using the same weighted digraph, but it can use negative values on the edges to calculate the result.

Unfortunately, it has a higher time complexity coming in at O(VE) but an advantage is it can also detect negative cycles if it is iterated once more past the point the values don’t change. A negative cycle occurs when a path’s edges sum to a negative value, which means in theory it could infinitely loop around the negative edge to become infinitely negative. Its operation starts by assuming the distances to all vertices are infinite in relation to a chosen source vertex, which has a value of zero. Then from the source, the edges relax, which means that paths of one edge from the source are investigated and the value which the edge stores to get to the next node is stored by the vertex. This happens iteratively, meaning that paths of two edges are now eligible and the shortest path found to any particular vertex is stored by it. This process happens at most V-1 times. That is to say, the number of vertices – 1.

In order to prevent the algorithm getting stuck in a perpetual negative cycle, logic was applied to prevent the path going back on itself by implementing a visited cities array “nodeCitiesVisitedIndexArray” which stored the index of the city just visited. I felt the use of an array here was justified as the list of unique cities being referenced can be adequately accessed by index, and since I didn’t see the need to store the strings in the same array as I could access them with pointers with the index, for example a matrix wasn’t needed.

**Testing**

In the labs, I learned that separating the program into source files and having a header file as an interface instead of putting all functions in the main file was a good way to segment the work and help with debugging. When I experienced issues with a new part of code I had added it was easier for me to see the general location of the problem.

Most of the testing conducted was the use of flags manifested as “printf()” statements inserted at key points in the function (just after a for loop or a function from another source file). I also implemented a “cityDisplay()” function which isn’t used in program’s operation but it allowed me to debug the import functions and how they were transferring the data into the relevant double linked lists.

In terms of things I should have done to make the program better, I would say that I should have implemented a function that could display negative cycle information to the user. I also should have included more code to deal with bad inputs as I imagine it would crash if they were not what I had programmed the functions to receive in some cases.

Shown below is the imported energy file (figure 1) and the output (figure 2).

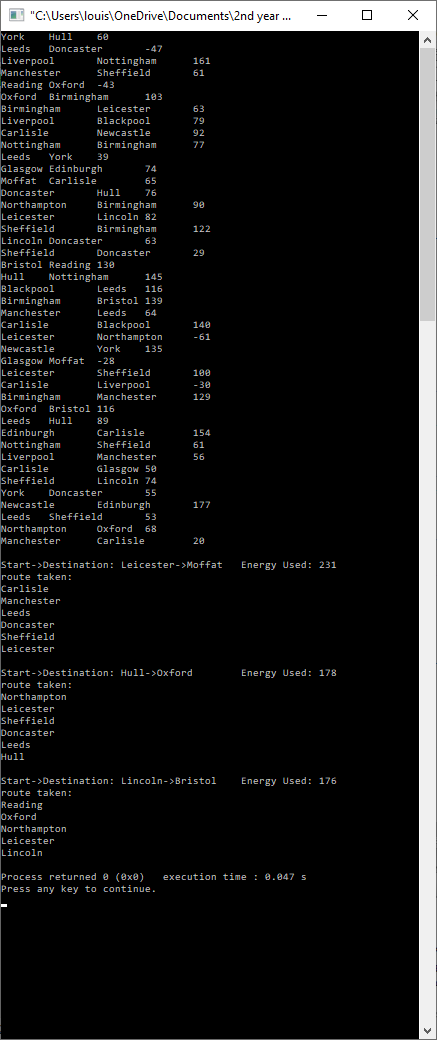
Figure 1


Figure 2 – output

Figure 1 – imported energy list