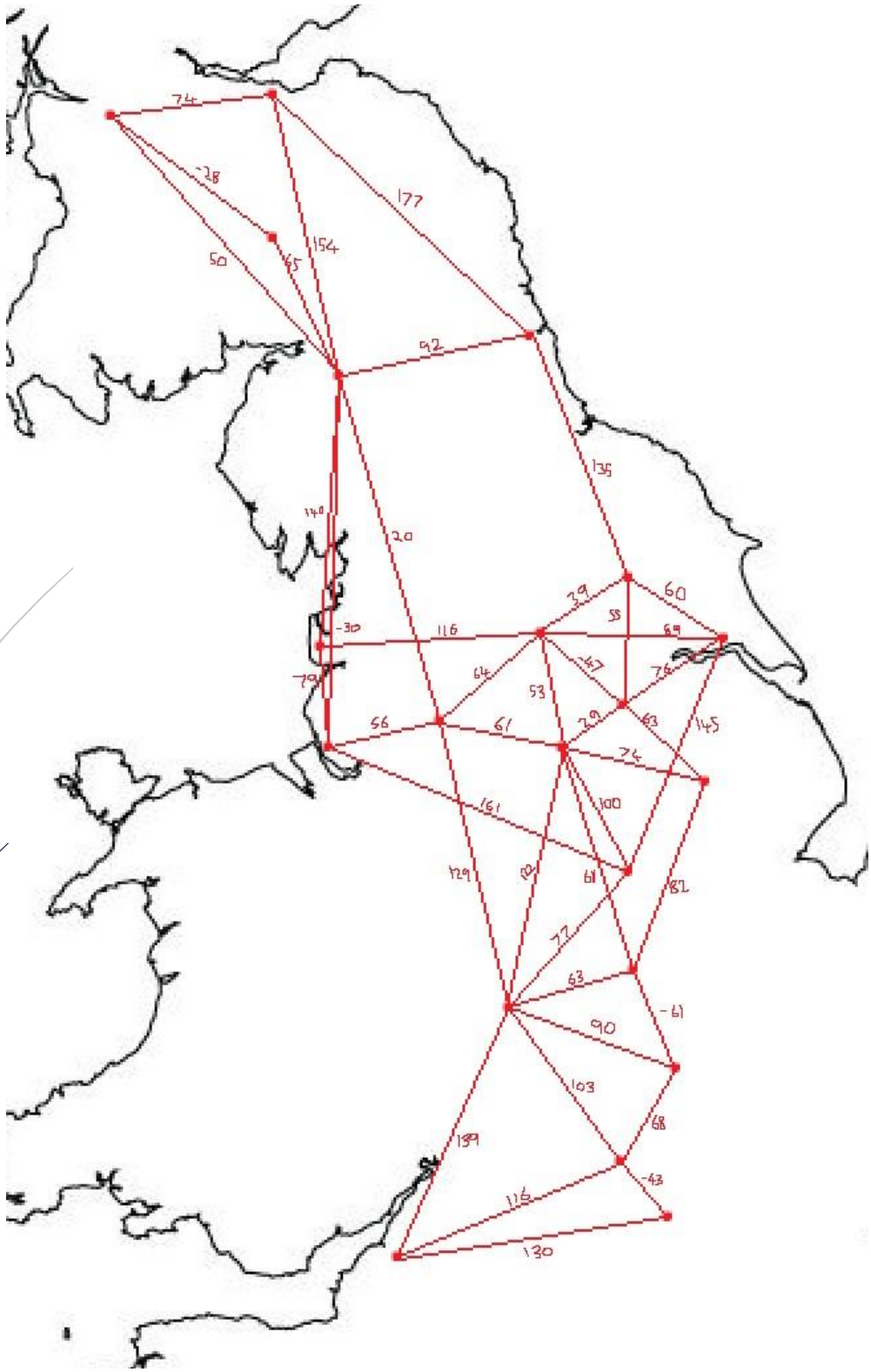
Exam No: Y3868917

CityPairs Project

Algorithms & Numerical Methods



1 // Abstract

The following report will document the development of a program aimed to assist a user of an electric vehicle in choosing the path of least energy expenditure from a source city to a given destination. The chargers implemented under the roads differ in efficiency, and so it is important that an optimal route can be decided, as to improve overall costs. Below will be a discussion on all stages of the development of this program, beginning with problem analysis. Descriptions of any problems encountered, and any methods used for resolving them will take place here. Limitations of said problems will also be specified. The next section goes over technical specifications of the program, and on each aspect of the code itself. This includes flow charts and my choice of algorithms in solving the shortest path problem. The final section is evaluative and will therefore critically analyse the entire code. Source code is included in the appendix for reference.

2 // Problem Description and Analysis

The most significant problem I encountered when working through this task was that I wanted a way to index each unique city, primarily so that I could call it as a integer in an array, such as:

```
for(int i=0; i<cityPairLineCount; i++){
    fscanf(fcitypairs, "%s\t%s", CityPair1,CityPair2);

    for(int j=0; j<CityCount; j++){
        if(strcmp(CityPair1, getName(j)) == 0){
            startCities[i] = j;
            break;
        }
    }
}
```

In this case, I was checking if the current city of the pair being assessed, ‘CityPair1’ is identical to the one currently stored in getName(j). This will be covered in more detail in section 3 of the report, but as can be shown I wanted to have j represent an index of a known city, and then for it to be assigned into the ‘startCities[]’ array at position i.

The solution I eventually came up with made use of the sprintf() function, as well as the function mentioned previously that I created, getName().

```
char *knownCitiesList;
knownCitiesList = malloc(128 * LineCount * sizeof(int));

char *getName(int c){
    char* name = knownCitiesList + c*128;
    return name;
}
```

This function operates on the fact that the character pointer ‘knownCitiesList’ has previously had 128 integers worth of memory assigned per line, and therefore for every value of ‘c’, it will be accessing different sections of memory where the names of the cities will be stored.

This was done using the following code:

```
if(!DuplicateCity1){
	sprintf(getName(CityCount), "%s", City1);
	CityCount++;
}
```

As stated before, I used the sprintf() function essentially copy the string passed by ‘City1’ into the memory location accessed by getName(CityCount), this was done every time a duplicate was not found upon reading the list, and CityCount would then be incremented by 1, going to the next location in getName(). These two methods resulted in a list of 21 unique cities in the memory that could now be used with integers rather than having to work around strings.

3 // Technical Specification

There were essentially three stages to this task in my eyes which I would have to overcome, which were as follows:

- Reading individual lines and subsequently single words from two given text documents.
- Creating a graph abstract data type, and the edges that would go inside it.
- Implementing an algorithm that would discover the path of least total weight between 21 different cities with different connections to each.

For this reason I will separate this section into three parts.

Reading Files

This was relatively simple to solve on its own, as I simply used the fopen() function to call a text document from my source folder, and make sure it was in read only mode. I also added a contingency in case this failed for some reason, and had the main function return -1.

```
FILE *fenergy = fopen("src/energy.txt","r");
FILE *fcitypairs = fopen("src/citypairs.txt","r");
if(fenergy == NULL || fcitypairs == NULL){
	printf("Error reading data file\n");
	return -1;
}
```

I next created a while loop utilizing the function fgets(), which reads characters from stream and stores them as a C string until a newline or the end-of-file is reached, and incremented the ‘LineCount’ variable each time it loops.

To then isolate individual words from each line, I used the function fscanf(), which reads data from the stream and stores it according to the parameter format into the locations pointed by the additional arguments, which in this case were City1, City2, and &energy. The reason the first two arguments do not need the referencing operator (&) before them is because these arguments are pointers to allocated storage, and since City1 and City2 are strings, they do not need to be referenced.

```
int DuplicateCity1 = 0;
int DuplicateCity2 = 0;
for(int j=0; j<CityCount; j++){
    if(strcmp(City1,getName(j)) == 0){
        DuplicateCity1 = 1;
    }
    if(strcmp(City2,getName(j)) == 0){
        DuplicateCity2 = 1;
    }
}
```

The next step was to determine if either city on the line had been previously read before, and if so not add it to the list of known cities. For this the strcmp() function was incorporated, which works by comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. If the strings are exactly alike the function will return zero. The rest of the task is just up to then passing the unique cities into the memory, and giving each of them an integer index, which is the step I covered in section 2.

Creating a Graph

The graph was done by taking what I had learnt in the labs on array lists and modifying it into a suitable structure for this task, in this case it was in the form of two separate structures, one for the main graph, which contains the total number of vertices and edges, as well as a pointer to an edge structure. This contains the source and destination city's indexes, as well as the energy expenditure to travel between them.

The function createGraph() was also modified from functions in the labs used to create lists.

```
struct Graph* createGraph(int E){
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->edge = malloc(E * sizeof(struct Edge));


    graph->vertices = MAX_CITIES; //
    graph->edgeNum = E; //

    return graph;
}
```

It begins with allocating memory for the graph structure, as well as enough for the total number of edges, E, which is set via the argument when called in the main function. The number of vertices has been given by MAX_CITIES, which is defined in the header as 21.

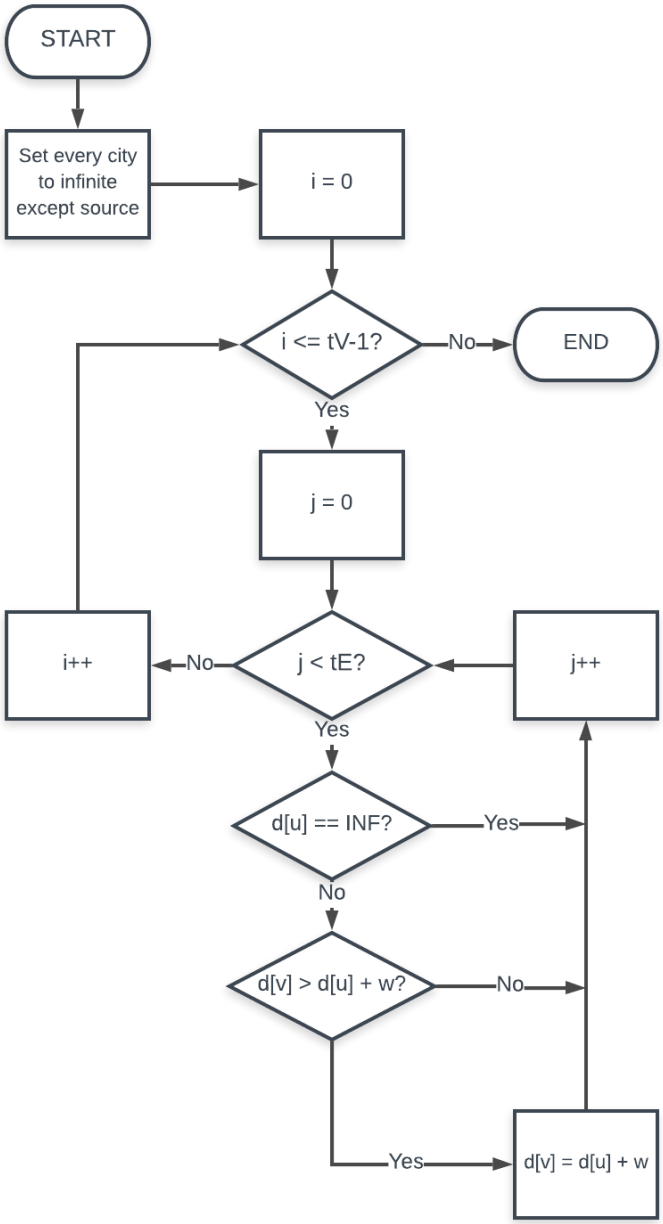
```
void addEdge(struct Graph* graph, int src, int dest, int wgt, int i){

    graph->edge[i].source = src;
    graph->edge[i].destination = dest;
    graph->edge[i].energy = wgt;
};
```

I then created a function to add an edge between two connected cities, which takes the graph, the source, the destination, the energy, and the i value of the loop the function is being called in. The purpose it serves is relatively simple, as it just assigns the argument values into the appropriate edge structure.

Implementing an Algorithm

With choosing an how to implement a shortest path algorithm, the two specific ones that came to mind were Bellman Ford's and Djisktra's. I ended up going with Bellman Ford's as it has the advantage of being able to work with negative weights, which we had in our energy.txt file. The way this algorithm works is it first calculates the shortest distances which have at most one edge in the path. Then it calculates shortest paths with at most 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum tV-1 edges in any simple path, that is why the outer loop runs tV-1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edge. The flowchart to the left shows the way in which I implemented this theory into my code.



4 // Testing

```
Vertex          Distance from Leicester
York                    345
Hull                    285
Leeds                   238
Doncaster                       361
Liverpool                       301
Nottingham                      140
Manchester                      246
Sheffield                       185
Reading                 123
Oxford                  166
Birmingham                      63
Leicester                       0
Blackpool                       354
Carlisle                        271
Newcastle                       480
Glasgow                 308
Edinburgh                       425
Moffat                  336
Northampton                     153
Lincoln                 424
Bristol                 253

Lowest amount of energy consumed between Leicester and Moffat is 336


Vertex          Distance from Hull
York                    60
Hull                    0
Leeds                   29
Doncaster                       76
Liverpool                       149
Nottingham                      166
Manchester                      93
Sheffield                       105
Reading                 282
Oxford                  325
Birmingham                      222
Leicester                       205
Blackpool                       145
Carlisle                        119
Newcastle                       195
Glasgow                 156
Edinburgh                       273
Moffat                  184
Northampton                     312
Lincoln                 139
Bristol                 412

Lowest amount of energy consumed between Hull and Oxford is 325


Vertex          Distance from Lincoln
York                    340
Hull                    280
Leeds                   127
Doncaster                       356
Liverpool                       191
Nottingham                      135
Manchester                      135
Sheffield                       74
Reading                 205
Oxford                  248
Birmingham                      145
Leicester                       82
Blackpool                       243
Carlisle                        161
Newcastle                       475
Glasgow                 198
Edinburgh                       315
Moffat                  226
Northampton                     235
Lincoln                 0
Bristol                 335

Lowest amount of energy consumed between Lincoln and Bristol is 335
```

To test the program, I modified the original energy.txt file and created a more compact version with a total of only 5 cities and 8 edges between them, as well as putting various printf() statements throughout the code to see how the variables would change throughout once compiled. I used this technique to make sure anything I was struggling with was working properly and if it was not, I was able to see where exactly the code was going wrong. Having fewer total cities also made it easier to walk through implementation of the Bellman Ford algorithm, as since it iterates only a maximum of tV-1 times, four total iterations was much more favourable than 20.

As can be seen from the screenshot of the console to the left, my final code was able to determine the most efficient routes from each of the source cities given in the citypairs.txt file, and then display the destination city and it's most energy effective route in its own statement at the end of each test.

5 // Evaluation

From talking to peers about the results they gathered, I am aware that my final values for the test pairs in the given citypairs.txt file are likely not correct. I could not determine why, as when I used a lower number of total cities the results were expected between them, so I can only assume that the implementation of the chosen algorithm is correct. Due to this I assume that the reason is either from there being negative energies between some cities causing an unknown error somewhere, or some unforeseen consequences of there being too many cities for the code to handle, although I could not figure out why.

Other than this, I do not see anything else particularly wrong with my code. I am sure there are more efficient ways to dynamically allocate memory to structures or stored variables, but I think that my method was more than sufficient.

If I were to attempt this project again, or any other project of a similar nature, I would possibly think about ways in which I could make the console output more user friendly or neater, or potentially add some sort of GUI, allowing users to choose the cities they want to be the source and destination while not having to alter a notepad.

Appendix

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "citypairs.h"

int main(){

	//reads both important files, plus prints an error if problems occur
	FILE *fenergy = fopen("src/energy.txt","r");
	FILE *fcitypairs = fopen("src/citypairs.txt","r");
	if(fenergy == NULL || fcitypairs == NULL){
		printf("Error reading data file\n");
		return -1;
	}

	//variables for processing fenergy file
	int CityCount = 0;
	int LineCount = 0;
	char line[1024];
    char City1[20];
	char City2[20];
	int Pair1;
	int Pair2;
	int energy;

	//reads each line and counts whenever a newline occurs
	while(fgets(line,1024,fenergy) != NULL){
		LineCount++;
	}
	//reset read position in document to start
    fseek(fenergy,0,SEEK_SET);

	//double LineCount as we need to have edges go both ways
    struct Graph* graph = createGraph(LineCount*2);

	knownCitiesList = malloc(128 * LineCount * sizeof(int));

	for(int i=0; i<LineCount; i++){
		//separating each word in each line
		fscanf(fenergy, "%s\t%s\t%d", City1,City2,&energy);

		//loops through the list to see if the city has already
		//occurred
		int DuplicateCity1 = 0;
		int DuplicateCity2 = 0;
		for(int j=0; j<CityCount; j++){
			if(strcmp(City1,getName(j)) == 0){
				DuplicateCity1 = 1;
			}
			if(strcmp(City2,getName(j)) == 0){
				DuplicateCity2 = 1;
			}
		}
```

```
        //if it hasn't, add it to the known cities list
        if(!DuplicateCity1){
              sprintf(getName(CityCount), "%s", City1);
              CityCount++;
        }
        //assign the city index to half of the pair
        for(int j=0; j<CityCount; j++){
              if(strcmp(City1,getName(j)) == 0){
                    Pair1 = j;
                    break;
              }
        }
        if(!DuplicateCity2){
              sprintf(getName(CityCount), "%s", City2);
              CityCount++;
        }
        for(int j=0; j<CityCount; j++){
              if(strcmp(City2,getName(j)) == 0){
                    Pair2 = j;
                    break;
              }
        }
        //add two edges with the pair just assigned
        //makes this bidirectional, with the same
        //energy gain/loss either way
    addEdge(graph, Pair1, Pair2, energy, i);
        addEdge(graph, Pair2, Pair1, energy, i);
  }

////////////////////////////

  //variables for processing fcitypairs file
  int cityPairLineCount = 0;
char CityPair1[20];
char CityPair2[20];

  //reads each line and counts whenever a newline occurs
  while(fgets(line,1024,fcitypairs) != NULL){
    cityPairLineCount++;
}
  //reset read position in document to start
fseek(fcitypairs,0,SEEK_SET);

  int* startCities = malloc(sizeof(int) * cityPairLineCount);
int* endCities   = malloc(sizeof(int) * cityPairLineCount);

for(int i=0; i<cityPairLineCount; i++){
        //separating each word in each line
    fscanf(fcitypairs, "%s\t%s", CityPair1,CityPair2);

        //assigns the startCities variable the index of the source city
    for(int j=0; j<CityCount; j++){
        if(strcmp(CityPair1, getName(j)) == 0){
            startCities[i] = j;
            break;
        }
    }

        //assigns the endCities variable the index of the destination
        //city
    for(int j=0; j<CityCount; j++){
```

```
            if(strcmp(CityPair2, getName(j)) == 0){
                endCities[i] = j;
                break;
            }
        }
            //run the main logic for each pair needed to be assessed
        BellmanFord(graph, startCities[i], endCities[i]);
    }

    return 0;
}
```

citypairs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "citypairs.h"

//integrates an index for the unique cities, which can then be called upon
//eg. in arrays
char *getName(int c){
    char* name = knownCitiesList + c*128;
    return name;
}

struct Edge{
    int source;
    int destination;
    int energy;
};

struct Graph{
    int vertices;
    int edgeNum;

    struct Edge* edge;
};

//function to create a graph
struct Graph* createGraph(int E){
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->edge = malloc(E * sizeof(struct Edge));


    graph->vertices = MAX_CITIES; //
    graph->edgeNum = E; //

    return graph;
}

//function to add an edge between two cities
void addEdge(struct Graph* graph, int src, int dest, int wgt, int i){

    graph->edge[i].source = src;
    graph->edge[i].destination = dest;
```

```
    graph->edge[i].energy = wgt;
};

//main logic to determine least energy consumed path
void BellmanFord(struct Graph* graph, int source, int destination){
    //variables
      int i;
      int j;
    int u;
    int v;
    int w;

      //total vertex number in the graph g
      int tV = graph->vertices;

      //total edge number in the graph g
      int tE = graph->edgeNum;

      //distance array
      //size equal to the number of vertices of the graph
      int d[tV];

      //fill the distance array with each being infinite
      for (i = 0; i < tV; i++) {
            d[i] = INF;
      }

      //mark the source vertex
      d[source] = 0;

      //main iterator, tV-1 times as this is the maximum possible
      for(i=1; i<=tV-1; i++) {
            for(j=0; j<tE; j++) {
                  //get the edge data
                  u = graph->edge[j].source;
                  v = graph->edge[j].destination;
                  w = graph->edge[j].energy;

                  if(d[u] != INF && d[v] > d[u] + w) {
                        d[v] = d[u] + w;
                  }
            }
      }

      //detect negative cycle
      //if value changes then we have a negative cycle in the graph
      //and we cannot find the shortest distances
      for(i=0; i<tE; i++) {
            u = graph->edge[i].source;
            v = graph->edge[i].destination;
            w = graph->edge[i].energy;
            if(d[u] != INF && d[v] > d[u] + w) {
                  printf("Negative weight cycle detected\n");
                  return;
            }
      }

    printf("Vertex\t\tDistance from %s\n", getName(source));
      for(i = 0; i<tV; i++){
        printf("%s\t\t\t%d\n", getName(i), d[i]);
      }
```

```
    printf("\nLowest amount of energy consumed between %s and %s is
            %d\n\n\n", getName(source), getName(destination),
            d[destination]);

    return;
}
```

citypairs.h

```
#define INF 99999
#define MAX_CITIES 21

char *knownCitiesList;
char *getName(int c);

struct Edge;
struct Graph;

struct Graph* createGraph(int E);
void addEdge(struct Graph* graph, int src, int dest, int wgt, int i);
void BellmanFord(struct Graph* graph, int source, int destination);
```