Programming report

Mohammad Al Shakoush

2020

1 Problem description

On the hand of the Netherlands map representing the a rail network consisting of 12 stations. Where each two stations have a connection that has a certain time or "weight". A program is designed to manipulate a certain query to find the shortest path between two stations. The input consists of four components:

- Number of disruptions
- Disrupted connections
- The starting station
- The goal station
- '!' indicating end of query

Figure 1: map of rail network

For each query the algorithm of Dijkstra is used to find the fastest connection then outputs the list of all stations along the shortest route, including the starting and ending station. In addition, the total time this route will take. In case the station is not reachable due to disruption the program should output "UNREACHABLE".
Example input-output behaviour :
2
Utrecht
Zwolle
Enschede
Zwolle
Amsterdam
Groningen
Enschede
Eindhoven

!
Amsterdam
Utrecht
Eindhoven
Nijmegen
Zwolle
Meppel
Groningen
269
UNREACHABLE

2 Problem analysis

Designing such a program requires dividing the task at hand into smaller sub-tasks. The first essential tasks are:

- Making the graph
- Reading input
- Traversing the graph via Dijkstra's algorithm
- Outputting the result

Each one of these tasks can be broken into smaller tasks:

1. Making the graph
Firstly, the graph vertices and edges:
Since we are only bound to traverse the rail-network of the Netherlands, hard-coding the graph is not a bad idea. However, hard-coding a graph is bad practice and leads to a static program that can strictly be used for one graph.
Therefore, making the graph with the help of a text file, so the user can save the names of connections with their weight into a text file, seems the most optimal way to make the program user friendly, dynamic and adaptive. An important upside to taking this approach is that the program does not have to be recompiled each time a (slight) modification of the graph is needed to be done.
An example of such a graph representation in a text file is as follows:

```
<stations name>
<stations name>
<cost of connection>
:
Amsterdam
Den Haag
46
```

This form of input makes the most sense, since you can put in the two names of the stations followed by the cost of their connection.
So the program should be able to make a new vertex if it does not already exist. This way the user does not have to first introduce the names of the stations to the program and then their connections. **Lastly**, constructing the graph with good data structures.
We can use a matrix, which stores edges that do and do not exists. However, a lot of memory is wasted this way. Therefore, we need to use a better data structure.

2. Reading input

Firstly, we need to read the number of disrupted stations. Followed by the the two stations where the disruption(s) occurred. Then remove the edges between the two vertices. We then need to repeat the process as many times as the number of disrupted stations.

Secondly, we need to read the query. Where two stations are called by name.

Because in this step and the previous one a string is scanned, we need to find the vertex with that string.

Lastly, we need to stop when the exclamation mark is read. Otherwise, read another query.

3. Traversing the graph via Dijkstra's algorithm

After removing the connection between the stations where the disruption occurred. We traverse the graph via Dijkstra's algorithm. We use the algorithm to find the fastest route to get from station A to station B. Stations A and B need to read from input and both vertices have to get recognized in the graph.

4. Outputting the result

The output consists of the stations visited along side the shortest path. Followed by time it takes to go through the particular path.
If no '!' is read expect another query.

3 Program design

Since we are dealing with a graph we need 3 main components:
1. Graph: contains all vertices
2. Vertex: has a name and neighbour vertices.
3. Edge: represents the connection between two vertices and the weight of that connection.

A **Graph** is an array of pointers to vertices.

A **Vertex** has multiple entities :

1. name: a string.
2. from: points to the vertex where we came from.
3. neighbour: points to a linked list of edges.
4. distance: integer to represent the weight of all edges we have been through up until goal
5. visited: Boolean; whether the vertex has been visited.

An **Edge** has three entities:

1. vertex: points the vertex connected to.
2. weight: holds the weight of the edge.
3. neighbour: points to the next edge.

The data structures used are represented in **figure 2**.

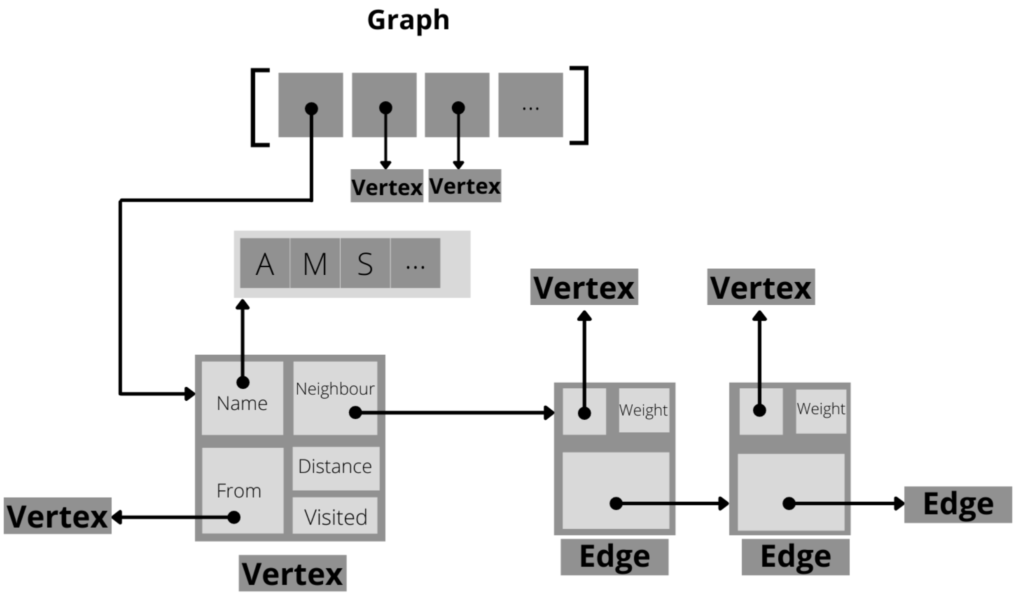


Figure 2: complete graph representation

To save memory I have chosen to represent the edges as linked lists. The process starts off with constructing the graph from the text file.
We do so by first making the data structure "Graph". Which is an array of vertices pointers.
We then start by scanning all sub-graphs. A sub-graph is the combination of a start vertex, end vertex and the weight of their connection.
We need to make the sub-graph by making two vertices each of which containing the name scanned. Unless the vertex already exists in our graph
After doing that we scan the weight of the connection and make an edge connecting start station with end station and vice versa, since we are dealing with a bidirectional weighted graph.

This is done by the function **makeSubGraph**, which is getting called by the function **makeNetwork**. **makeNetwork** moves the file pointer forward and calls **makeSubGraph** until **E**nd **O**f **F**ile is reached.

After constructing the graph. We start with **scanning input from user;**

Firstly, we scan the number of disruption.
Secondly, we scan the names of the start and end vertices where the edge must be removed. This is done with the function **removeEdge** which removes an edge in the linked list. After all edges where disruption(s) occurred are removed from the graph. We start by **scanning the query**.
We scan the query by scanning the start and goal strings. Once the strings are scanned we look for the vertices with those names in the graph. This is manily done by **scanVertex** function
Now, all we have to do is call **dijkstra**. Before starting wiht dijkstra, the following entites are set to the following values :
visited = **FALSE**
from = **NULL**
distance = **infinity**
This is done to all vertices when making them. Then **Dijkstra** starts at the start vertex. Where we set the distance to 0. As long as we have a not visited vertex we keep iterating. With each iteration we choose the closest station that has not been visited yet. Which is determined by the entity **distance**, **visited** of the vertex. Then we update the distance of all neighbors. The neighbours distance is only updated if the new distance found is less than the already discovered one (the value of the entity **distance**). So, with each iteration the distance is summed with the previous one. So once we reach the goal the total distance is calculated. In addition to the distance we update the entity **from** to point to the current vertex.

Starting at the goal vertex we can go back using the entity **from** to the vertex we visited previously. Until start vertex is reached, where **from** points to NULL. This is handled with the function **printRoute**. Unless the goal vertex still has a distance of infinity indicating it is unreachable which in that case UNREACHABLE is printed. The time is **distance** at the goal vertex.
After printing the route and the time. We call the function **eraseRoute** which resets all vertices of the graph by resetting the three entities **distance, visited, from** preparing the graph for a new traverse.

This process repeat until '!' is read. Then the graph gets freed by **freeGraph** function.

4 Evaluation of the program

The program performs very well. After all, it passed all test cases on Themis.
The text file containing the connection was as follows:

```
Amsterdam
Den Haag
46
Amsterdam
Den Helder
77
Amsterdam
Utrecht
26
Den Haag
Eindhoven
89
Eindhoven
Maastricht
63
Eindhoven
Nijmegen
55
Eindhoven
Utrecht
47
Enschede
Zwolle
50
Groningen
Leeuwarden
34
Groningen
Meppel
49
Leeuwarden
Meppel
40
Maastricht
Nijmegen
111
Meppel
Zwolle
15
Nijmegen
Zwolle
77
Utrecht
Zwolle
51
```

The test case used is the first one from Themis:

```
2
Utrecht
Zwolle
Enschede
Zwolle
Amsterdam
Groningen
Enschede
Eindhoven
Leeuwarden
Eindhoven
!
```

The output is the following:

```
Amsterdam
Utrecht
Eindhoven
Nijmegen
Zwolle
Meppel
Groningen
269
UNREACHABLE
Leeuwarden
Meppel
Zwolle
Nijmegen
Eindhoven
187
```

When testing the program with Valgrind, the following is yielded.

```
[redacted]:~/gitHub/CSyear1/ADT/assignments/5Trains$ make debug
gcc -O2 -std=c99 -pedantic -Wall -Wno-unused-result main.c libGraph.c dijkstra.c -g -o main
valgrind --error-exitcode=111 --leak-check=full --track-origins=yes ./main < testCase1.txt
==558== Memcheck, a memory error detector
==558== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==558== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==558== Command: ./main
==558==
==558== error calling PR_SET_PTRACER, vgdb might block
Amsterdam
Utrecht
Eindhoven
Nijmegen
Zwolle
Meppel
Groningen
269
UNREACHABLE
Leeuwarden
Meppel
Zwolle
Nijmegen
Eindhoven
187
==558==
==558== HEAP SUMMARY:
==558==     in use at exit: 0 bytes in 0 blocks
==558==   total heap usage: 89 allocs, 89 frees, 14,564 bytes allocated
==558==
==558== All heap blocks were freed -- no leaks are possible
==558==
==558== For lists of detected and suppressed errors, rerun with: -s
==558== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 3: test for memory leak

Indicating no memory leaks or any illegal memory access has been detected.

5 Extension of the program

As mentioned earlier, the extension of the program is that the user can use the same program for other (larger) networks. This can be easily done by adding the stations and their connections in the text file.
Two macros used called **MAXSTRING** and **MAXSIZE** are initialy defined for the Netherlands network. However, while reading the text file the program reallocates enough memory for any size of networks and any size of a string.
This is implemented in the **makeGraph** and **scanStringFromConsole** functions.
An example of a rail network between multiple countries and many stations is tested. The following stations are added to the Netherlands rail network

```
Berlin
Munich
240
Berlin
Frankfurt
250
Berlin
Hamburg
105
Berlin
Cologne
270
Hamburg
Frankfurt
270
Hamburg
```

```
Cologne
270
Hamburg
Munich
360
Frankfurt
Munich
218
Frankfurt
Cologne
75
Munich
Cologne
275
Paris
Frankfurt
150
Paris
Brussel
50
London
Amsterdam
120
London
Frankfurt
350
Dublin
London
120
```

Stations add are in Germany, France and UK. These stations are also linked to the one's in the Netherlands making the graph bigger and connecting international stations

Test input :

```
0
Paris
Dublin
Hamburg
Amsterdam
Dublin
Munich
Frankfurt
Groningen
London
Den Haag
Cologne
Enschede
!
```

The program outputs the following:

```
Paris
Frankfurt
London
Dublin
620
Hamburg
Berlin
```

```
Amsterdam
315
Dublin
London
Frankfurt
Munich
688
Frankfurt
Berlin
Amsterdam
Utrecht
Zwolle
Meppel
Groningen
601
London
Amsterdam
Den Haag
166
Cologne
Berlin
Amsterdam
Utrecht
Zwolle
Enschede
607
```

Valgrind returns 0 errors.

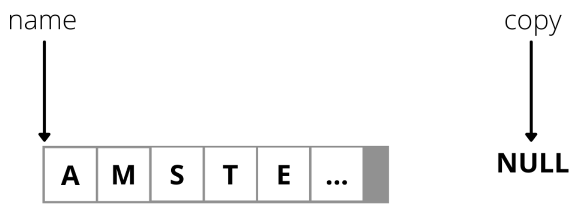
To ensure the graph is actually made. I wrote a **printGraph** function that can be used to print the graph on screen. This is the graph we made with the previous file :

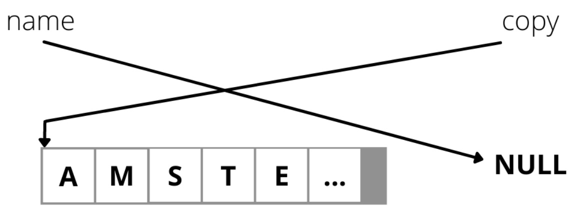
```
Berlin
 to ->  Hamburg : 105.
 to ->  Amsterdam : 210.
 to ->  Munich : 240.
 to ->  Frankfurt : 250.
 to ->  Cologne : 270.

Munich
 to ->  Frankfurt : 218.
 to ->  Berlin : 240.
 to ->  Cologne : 275.
 to ->  Hamburg : 360.

Frankfurt
 to ->  Cologne : 75.
 to ->  Paris : 150.
 to ->  Munich : 218.
 to ->  Berlin : 250.
 to ->  Hamburg : 270.
 to ->  London : 350.

Hamburg
 to ->  Berlin : 105.
 to ->  Cologne : 270.
 to ->  Frankfurt : 270.
 to ->  Munich : 360.

Cologne
 to ->  Frankfurt : 75.
 to ->  Hamburg : 270.
 to ->  Berlin : 270.
 to ->  Munich : 275.

Paris
 to ->  Brussel : 50.
 to ->  Frankfurt : 150.

Brussel
 to ->  Paris : 50.

London
 to ->  Dublin : 120.
 to ->  Amsterdam : 120.
 to ->  Frankfurt : 350.

Amsterdam
 to ->  Utrecht : 26.
 to ->  Den Haag : 46.
 to ->  Den Helder : 77.
 to ->  London : 120.
 to ->  Berlin : 210.

Dublin
 to ->  London : 120.

Den Haag
 to ->  Amsterdam : 46.
 to ->  Eindhoven : 89.

Den Helder
 to ->  Amsterdam : 77.

Utrecht
 to ->  Amsterdam : 26.
 to ->  Eindhoven : 47.
 to ->  Zwolle : 51.

Eindhoven
 to ->  Utrecht : 47.
 to ->  Nijmegen : 55.
 to ->  Maastricht : 63.
 to ->  Den Haag : 89.

Maastricht
 to ->  Eindhoven : 63.
 to ->  Nijmegen : 111.

Nijmegen
 to ->  Eindhoven : 55.
 to ->  Zwolle : 77.
 to ->  Maastricht : 111.

Enschede
 to ->  Zwolle : 50.

Zwolle
 to ->  Meppel : 15.
 to ->  Enschede : 50.
 to ->  Utrecht : 51.
 to ->  Nijmegen : 77.

Groningen
 to ->  Leeuwarden : 34.
 to ->  Meppel : 49.

Leeuwarden
 to ->  Groningen : 34.
 to ->  Meppel : 40.

Meppel
 to ->  Zwolle : 15.
 to ->  Leeuwarden : 40.
 to ->  Groningen : 49.
```

Figure 4: output of printGraph

6 Process description

Mohammad (me) programmed everything and made the report.
What I have learned:
I have only heard about file pointers, but not actually programmed anything with them. So with this assignment I learned a lot about file pointers and how they operate.
I learned a lot about Dijkstra's algorithm and how to implement it. In addition to the down side of using it for huge networks.
This is also my first time making my own data structure. So I learned how to make them and sketch those to make them work optimally.
One thing I came across is the fact that I could ditch the **strcpy** function.
What I did instead is exchanging the pointers. As seen in **libGraph.c** lines 17,18. Also represented in the following figure.





7 Conclusions

The program solves the problem and is dynamic enough to be used for any kind of rail network, no matter the length of the names or how many stations it the network has. This also is done without even recompiling the whole program each time the rail network need to be modified.
The program can be improved by improving the function **closestStation**.
This function now iterates through the whole graph to find the closest station. The program can be upgraded to use a linked list or a min-heap where the least distance is at first position making the search for the least distance of $O(1)$ instead of iterating through the whole graph to find the closest station which is of $O(n)$ where n is the number of stations.
A down side of choosing to make the graph the way mentioned earlier is time complexity. Because the program has to check whether a stations already exists, if no makes one.
A slightly faster way would be introducing the names of all stations then their connections. This makes adding a station go from $O(n)$ where n is the number of stations already made to $O(1)$.

8 Appendix: program text

libGraph.h

```
#ifndef LIBGRAPH_H
#define LIBGRAPH_H

#define MAXSIZE 12 /*number of intital stations*/
#define MAXSTRING 11    /*characters in a name*/
/*the previous values are just as a start. Once an overflow happens a
   reallocation takes place*/
#define string char*    /*ease of reading*/
#define infinity 32767  /*max int value*/

typedef struct token *Edge;
typedef struct vertex *Vertex;
typedef struct graph *Graph;

#include <stdbool.h>

typedef struct vertex {
    string name;
    Edge neighbour; /*points to the head of the linked list*/
```

```c
    int distance; /*we keep track of distance to a vertex with modifying the
        weight of an edge*/
    Vertex from;    /*our track to the goal*/
    bool visited;
} vertex;

typedef struct token {
    Vertex vertex;
    int weight;
    Edge next;
} token;

typedef struct graph {
    Vertex *stations;
} graph;

void freeGraph(Graph G);
Graph newGraph ();
void makeStation (Graph G, string station);
void removeEdge (Vertex V, Vertex V1);
bool stationExists (Graph G, string name, int *i);
void makeEdge (Vertex station, Vertex neighbour, int weight);

#endif
```

main.c

```c
/**
 * Program made my Mohammad Al Shakoush. Last commit is on 28th of March
   2021.
 * The program reads a graph from a text file named "railNetwork.txt" and a
 * query of a start and end vertex and outputs the fastest way using
   dijkstra.
 * the program termiantes only when '!' is read.
 * Please use the make file included to make your life easier.
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "libGraph.h"
#include "dijkstra.h"

/*scans the string for input console*/
string scanStringFromConsole () {
    string name = calloc( (MAXSTRING+1), sizeof(char));
    assert(name!=NULL);
    int i = 0, size = MAXSTRING;
    char c = getchar();
    while (c != '\n' && c != '!') {
        if (i == size) {
            name = realloc(name, (size+10) * sizeof(char));
            size += 10; //update size
        }
        name[i] = c;
        i++;
        c = getchar();
    }
    return name;
}

```

```c
/*find the vertex with the same name and returns it*/
Vertex scanVertex (Graph railNetwork) {
    string name = scanStringFromConsole();
    if (name[0]=='!') {
        free(name);
        return NULL;
    }
    int i = 0;
    stationExists(railNetwork,name,&i);//i holds the index of the vertex
    free(name);
    return railNetwork->stations[i];
}

/*scans where disruption occurs and removes edges*/
void scanDisruptions (Graph railNetwork) {
    int disruptions;
    if(!scanf("%d\n", &disruptions)) {
        printf("disruptions not scanned correctly\n");
    }
    while (disruptions) {
        //scan 2 strings : start, end
        //delete the path from graph
        Vertex start, end;
        start = scanVertex(railNetwork);
        end = scanVertex(railNetwork);
        removeEdge(start,end);  //remove edge on both ends
        removeEdge(end,start);
        disruptions--;
    }
}

/*uses dijkstra's algorithm to traverse graph. After scanning start and end
    station*/
void traverseNetWork (Graph railNetwork) {
    scanDisruptions(railNetwork);

    int i = 0, j = 0, time = 0;
    Vertex start, goal;
    start = scanVertex (railNetwork);    //vertex where we begin
    if (!start) printf("start not scanned correctly\n");
    while (start) {
        time = i = j = 0;
        goal = scanVertex (railNetwork); //scans goal vertex
        time = dijkstra(railNetwork,start,goal);
        if (time == infinity) { //we have not visited goal station
            printf("UNREACHABLE\n");
        } else {
            printRoute(goal);   //prints route begining at goal tracing back
                the steps
            printf("%d\n", time);
        }
        eraseRoute(railNetwork);    //resets all modified values
        start = scanVertex(railNetwork);
    }
}

/*scans a line from the file character by character and returns the string*/
string scanLineFromFile (FILE *fp) {
    string name = calloc( (MAXSTRING+1), sizeof(char));
    assert(name!=NULL);
    char c = fgetc(fp);
```

```
    int i = 0 ;
    while (c != '\n') {
        if (i == MAXSTRING) {   /*if a station name is longer than 11(
            MAXSTRING) chars*/
            name = realloc(name,MAXSTRING+10 * sizeof(char));
        }
        name[i] = c;
        c = fgetc(fp);
        i++;
    }
    return name;
}

/*looks for the vertex in the graph with the same name. If that vertex does
    not exists
 *then make a new one.*/
Vertex stringToVertex (FILE *fp, Graph network) {
    string name = scanLineFromFile(fp);
    int i = 0;
    if (!stationExists(network, name, &i)) { /*check if station exists*/
        makeStation(network, name);
    } else {
        free(name);
    }
    return network->stations[i];
}

/*makes a subgraph of 2 vertices and one edge*/
void makeSubGraph (FILE *fp, Graph network) {
    Vertex start= stringToVertex(fp,network);
    Vertex end = stringToVertex(fp,network);
    /*next line is the weight of the edge with a new line char*/
    int weight = 0;
    fscanf(fp, "%d\n", &weight);
    /*here we have 2 vertices with an edge weight to connect*/
    makeEdge(start, end, weight);   //make edge on both ends
    makeEdge(end, start, weight);
}

/*construcs a network*/
void makeNetwork (Graph network) {
    FILE *fp;
    /*File pointer to read the file with the edges and vertices to make*/
    fp = fopen("railNetwork.txt", "r");
    /*while loop scans name and makes the graph*/
    fpos_t pos; fgetpos(fp, &pos);
    while (!feof(fp)) { /*end of file of fp*/
        fsetpos(fp, &pos); //assign a position for file pointer (fgetc
            modifies the pointer position)
        /*makes a sub graph from file (vertex, vertex, edge)*/
        makeSubGraph(fp, network);

        fgetpos(fp, &pos); //backs up the file pointer position
        fgetc(fp);
    }
    fclose(fp);
}

int main(int argc, const char **argv) {
    Graph railNetwork = newGraph();
    makeNetwork(railNetwork);
```

```
    traverseNetWork(railNetwork);
    freeGraph(railNetwork);
}
```

libGraph.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "libGraph.h"

Graph newGraph () {
    Graph G = malloc(sizeof (struct graph));
    assert(G!=NULL);
    G->stations = calloc( (MAXSIZE+1), sizeof (Vertex)); /* +1 to indicate the end (NULL)*/
    assert(G->stations!=NULL);
    return G;
}

void makeStation (Graph G, string station) {
    static int i = 0; static int size = MAXSIZE;    //static to keep their values
    string copy = station;  //we exchange the pointers instead of using strcpy
    station = NULL; //make initial pointer point to NULL to not change copy
    Vertex temp = calloc(1,sizeof(struct vertex));  //calloc takes care of initialization other entities to NULL
    assert(temp!=NULL);
    temp->name = copy;  //assign name of station
    temp->distance = infinity;
    /*for if stations > 12 */
    if (i == size) {
        size = size * 2;
        G->stations = realloc(G->stations, (size+1) * sizeof (Vertex));
    }
    G->stations[i] = temp; //add station to railNetwork aka graph
    i++;
}

/*checks whether a station exists and i holds the index
    of the pointer pointing to that station*/
bool stationExists (Graph G, string name, int *i) {
    while (G->stations[(*i)] != NULL) {
        if (!strcmp(G->stations[(*i)]->name, name)) { /*returns 0 if name == name*/
            return true;
        }
        (*i)++;
    }
    return false;
}

Edge makeNode (Vertex neighbour, int weight) {
    Edge temp = malloc(sizeof(struct token));
    assert(temp!=NULL);
    temp->vertex = neighbour;
    temp->weight = weight;
    temp->next = NULL;
    return temp;
```

```c
}

/*makes an edge. Puts it in ascending order*/
void makeEdge (Vertex station, Vertex neighbour, int weight) {
    Edge E = makeNode(neighbour, weight);
    /*start of linked list*/
    if ((station)->neighbour == NULL || weight <= (station)->neighbour->
        weight) {
         E->next = (station)->neighbour;
         (station)->neighbour = E;
    } else {    //not first node
        Edge temp = (station)->neighbour;
        /*we keep looping until a value is higher*/
        while(temp->next != NULL && temp->next->weight < weight) {
            temp = temp->next;
        }
        E->next = temp->next;
        temp->next = E;
    }
}

/*removes an edge aka disruption*/
void removeEdge (Vertex V, Vertex V1) {
    Edge E = V->neighbour;
    if (E->vertex == V1) {  //first edge
        V->neighbour = V->neighbour->next;
        free(E);
        return;
    }
    while(E->next && E->next->vertex != V1) {
        E = E->next;
    }
    Edge E1 = E->next;
    E->next = E->next->next;
    free(E1);
}

/*prints graph in a tree like way*/
void printGraph (Graph G) {
    for (int i=0; G->stations[i] != NULL; i++) {
        printf("%s", G->stations[i]->name);
        Edge E = G->stations[i]->neighbour;
        while (E != NULL) {
            printf("\n to ->  %s", E->vertex->name);
            printf(" : %d.", E->weight);
            E = E->next;
        }
        printf("\n\n");
    }
}

/*frees edges backwards*/
void freeN (Edge E) {
    if (E == NULL) {
        return;
    }
    freeN(E->next);
    free(E);
}

/*frees graph*/
```

```
void freeGraph(Graph G) {
    for (int i=0;G->stations[i] != NULL;i++) {
        freeN(G->stations[i]->neighbour);
        free(G->stations[i]->name);
        free(G->stations[i]);
    }
    free(G->stations);
    free(G);
}
```

dijkstra.h

```
#ifndef DIJKSTRA_H
#define DIJKSTRA_H

#include "libGraph.h"

Vertex closestStation(Graph railNetwork);
void eraseRoute (Graph G);
void printRoute (Vertex V);
int dijkstra (Graph railNetwork, Vertex start, Vertex goal);

#endif
```

dijkstra.c

```
#include <stdio.h>
#include <stdlib.h>
#include "libGraph.h"

/*goes back until from goal to start. Then prints*/
void printRoute (Vertex visited) {
    if (visited == NULL) {
        return;
    }
    printRoute(visited->from);
    printf("%s\n", visited->name);
}

/*resets the values modifed to erase the route for next traverse*/
void eraseRoute (Graph railNetwork) {
    int i = 0;
    Vertex temp = railNetwork->stations[i];
    while(temp!=NULL) {
        railNetwork->stations[i]->distance = infinity;
        railNetwork->stations[i]->visited = false;
        railNetwork->stations[i]->from = NULL;
        i++;
        temp = railNetwork->stations[i];
    }
}

/*returns the closest station*/
Vertex closestStation(Graph railNetwork) {
    int i = 0;
    int distance = infinity;
    Vertex V = railNetwork->stations[i];
    Vertex temp = railNetwork->stations[i];
    while (temp!=NULL) {
        if (!temp->visited &&
            temp->distance < distance) {
```

```
            distance =temp->distance;
            V = temp;
        }
        i++;
        temp = railNetwork->stations[i];
    }
    return V;
}

/*returns the number of vertices in a graph*/
int nrVertex (Graph railNetwork) {
    int i = 0;
    while(railNetwork->stations[i]) {
        i++;
    }
    return i;
}

int dijkstra (Graph railNetwork, Vertex start, Vertex goal) {
    start->distance = 0; //set initail distance to 0
    Vertex current;

    int nrUnvisited = nrVertex(railNetwork);  //the number of unvisited
       stations
    while (nrUnvisited) {
        current = closestStation(railNetwork);  //we start at closest
           station aka least distance
        Edge neighbors = current->neighbour;
        /*update the distance for the neighbors of the current vertex*/
        while(neighbors != NULL) {
            int weight = current->distance; //weight of edge
            if (neighbors->vertex->distance > neighbors->weight + weight
                && !neighbors->vertex->visited){    //we only update
                   distance when it is less than already found one
                neighbors->vertex->distance = neighbors->weight + weight;
                neighbors->vertex->from = current;  //save where we came
                   here from
            }
            neighbors = neighbors->next;    //move on to next neighbour
        }
        current->visited = true;    //one less unvisited vertex
        if(goal->visited || current->distance == infinity) {    //we stop if
            goal vertex is visited
            return current->distance;                           //we also
               stop when the closest one can't be reached
        }
        nrUnvisited--;
    }
    return infinity;
}
```

make file

```
CC     = gcc
CFLAGS = -O2 -std=c99 -pedantic -Wall -Wno-unused-result

graph:main.c libGraph.c
  $(CC) $(CFLAGS) $^ -g -o $@

main: main.c libGraph.c dijkstra.c
  $(CC) $(CFLAGS) $^ -g -o $@
```

```
test: main
	./main < input.txt

debug: main
	valgrind --error-exitcode=111 --leak-check=full --track-origins=yes ./main < input.txt

time: main
	time ./main < input.txt

clean:
	rm main
	clear
```