Assignment 5: Trains
Programming report

s4296850 & s4740556

Algorithms and Data Structures 2022

1 Problem description

General: We need to find a way between train stations in the Netherlands in an efficient manner. The program outputs the cities we have to go through and the time of the journey. However, because of bad weather, some connections between cities may be canceled. The program has to take that into consideration. There are some assumptions that simplify the task. We do not take into consideration the time needed to change trains and we do not wait for the trains. To put it in other words, the time given is the time spent commuting between the stations.

Input-output behavior: Input: First the user inputs the number n of disruptions. This is followed by $2n$ cities between which the connections are disrupted. A disruption between cities A and B means that travel from A to B and from B to A is not possible. After the disruptions are inputted, the user writes down pairs of the cities for which optimal connections are to be found. When the program is to be terminated, an exclamation mark should be written down. If at any point the user inputs a city that is not on the list of possible cities, the program exits.

Output: For each connection, the program outputs the list of cities visited from the starting point to the terminal station. Each city is printed in a new line. This is then followed by the time of the journey in minutes. Because of the disruptions, it is possible that some cities cannot be reached. Then, the program outputs "UNREACHABLE".

Example input:	Corresponding output:
`3`	`Utrecht`
`Zwolle`	`Eindhoven`
`Utrecht`	`Maastricht`
`Nijmegen`	`Nijmegen`
`Eindhoven`	`Zwolle`
`Amsterdam`	`Enschede`
`Den Helder`	`348`
`Utrecht`	`UNREACHABLE`
`Enschede`	`Meppel`
`Den Helder`	`Zwolle`
`Amsterdam`	`Nijmegen`
`Meppel`	`Maastricht`
`Eindhoven`	`Eindhoven`
`!`	`266`

2 Problem analysis

The problem can be divided into 3 sub-problems.
Initialisation of the graph. The program needs data about the network of the cities. It needs to know the name of the cities which have rail connections and it needs to know the time it takes to commute from one city to another. Although it is possible to hard code every possible city and connection between the cities into the program, such an approach does not allow any flexibility. This solution would work for the base problem, although would not work for real-life implementation or Extra 2 parts of the assignment. The other approach is to implement everything dynamically, by having a data file with the cities and the connections between them.

This option would be the best to use, as possible future changes are easy to implement. Our approach combines the two previously mentioned. We have hard-coded the number and names of the cites, while the connections between them are read from an external file. In a real-life scenario that file could be substituted with a database that would be easy to modify.

Implementing disruptions. After the program is initialized, the user can input disruptions between the cities. This part was rather easy to implement. Since we are storing connections between cities in a 2D matrix, we can just set the desired connections to have the value of 0, indicating there is no connection between the two cities.

Finding connections. This is the most complex part of the program. We need to find the optimal connection between the cities in an optimal manner. To achieve that we use Dijkstra's shortest path search algorithm. For extra efficiency, the program uses a heap structure which helps it access the closest cities first in an efficient manner.

3 Program design

Design choice. The design follows the three sub-problems listed above. The connections between the cities are stored in a 2D matrix of integers representing the time needed to travel from one city to another. The connections are read from an external text file. The program uses Dijkstra's shortest path search algorithm implemented using a heap structure. The program assigns a number to each of the cities, as it is easier to work on numbers than it is for names.

Matrix of Connections. Every city is assigned a number from 0 to 11, which makes a straightforward to implement the connections as a 2D array. A connection from city A to B is stored in row A, column B, and row B, column A. This is because the connections are always two-way connections. Using that structure allows us to easily find and modify the connections. It is not an optimal structure in terms of memory. However, for a small number of cities (12), it is not an issue.

Dijkstra's Algorithm. We were advised to use Dijkstra's Algorithm because it is regarded as the most efficient solution for weighted graphs. However, without the A* heuristic, it does not take geographical data into consideration. Therefore, although efficient enough for the assignment, it is not the best option available. Explanation of the algorithm is not in the scope of this report.

Heap in Dijkstra's Algorithm. In Dijkstra's Algorithm, the next node to access is chosen by a priority queue. The algorithm first explores the nodes which are closest in distance to the starting point. To efficiently implement this functionality into the algorithm, we have used a heap. The heap structure has a node with the smallest value on the top. New nodes are enqueued by their current distance to the starting node. This means we also have an array to keep track of which city this distance corresponds to, the number of which is enqueued as well. The city number is returned by function returnMin, which dequeues the node with the minimum value. We start the program by enqueuing the start node, we can terminate it if the heap is empty.

Time Complexity. Let n be the number of nodes and e the number of edges. The connections are added to the array one by one from the file, therefore $\mathcal{O}(n)$. The connections in the array can be directly accessed, with $\mathcal{O}(1)$. In Dijkstra's Algorithm itself, each node is added to the heap at maximum once before it is visited, resulting in a complexity of n (if a shorter distance to it is found, it is enqueued for a second time, however, if a node has been already visited and is dequeued from the heap for a second time, then it is just removed). Re-positioning in the heap can happen for every connection from any node, as when we add a new node we might need to upheap it up to the top. This can happen from every node for each edge, altogether $2e$, therefore $\mathcal{O}(e)$. Each node is removed from the heap maximum of 1 time, $\mathcal{O}(n)$. Therefore $2n + 2e$ actions, which is $\mathcal{O}(n + e)$. Each action in a heap is $\mathcal{O}(log(n))$, therefore the whole algorithm is $\mathcal{O}(log(n) * (n + e))$. All in all, we see that the whole program has a complexity of $\mathcal{O}(log(n) * (n + e))$, where n is the number of nodes and e the number of edges in the graph of connections.

4 Evaluation of the program

While developing the program, we were testing the performance on the example files given. Once the program was functional we used Valgrind to identify and resolve memory leaks, allocations of incorrect size, and accesses to uninitialized memory.

The final program was accepted by Themis and it passed all the test cases. It was tested for memory leaks by Valgrind. This helped to reveal, using an additional flag, that we were not closing the file containing connections.

After resolving this, there were no further leaks. On the input shown in the example above Valgrind generated the following output:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./a.out < input.txt
==5508== Memcheck, a memory error detector
==5508== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5508== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5508== Command: ./a.out
==5508==
Utrecht
Eindhoven
Maastricht
Nijmegen
Zwolle
Enschede
348
UNREACHABLE
Meppel
Zwolle
Nijmegen
Maastricht
Eindhoven
266
==5508==
==5508== HEAP SUMMARY:
==5508==     in use at exit: 0 bytes in 0 blocks
==5508==   total heap usage: 20 allocs, 20 frees, 9,824 bytes allocated
==5508==
==5508== All heap blocks were freed -- no leaks are possible
==5508==
==5508== For lists of detected and suppressed errors, rerun with: -s
==5508== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5 Process description

The development of the program was fruitful and rewarding. We first started by implementing the functionality of the program while using arrays for the connections and the search in Dijkstra's algorithm. It worked as a proof of concept, as it passed the first Themis case. Unfortunately, the method was too time inefficient. Afterward, we implemented a heap into our program. The heap was taken from the reader but had to be modified so that it is sorted for the smallest, not the highest value. Moreover, the array with numbers of cities was added. It was surprisingly easy and soon the program worked properly. We applied cosmetic changes, added comments, and submitted the code. The program was developed by us collaboratively during a part of a tutorial and then during a lab using a real-time collaborative environment available on `www.replit.com`. It is therefore impossible to distinguish which part of the program was written by who. Having more than a week until the deadline we have started working on the extras. They further increased the functionality and efficiency, but most importantly, we had a great amount of fun.

6 Conclusions

The program solves the problem of finding a path by using Dijkstra's Algorithm, which is regarded as the most efficient solution for weighted graphs without using heuristics. It can be optimized using the A* heuristic using geo-coordinates of the stations, which we have done in the extra exercise. In this implementation, however, the algorithm is not flexible, since we have defined the constant size of the 2D array and we are using fixed names of the cities. Now we store duplicated nodes in the queue, although skip them when they were already visited. The time complexity has been previously described. The flexibility of the program is increased for Extra 2, although it is not yet fully functional

7 Appendix: program text

Listing 1: trains.c

```c
/* name: trains.c
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 23.03.2022
 *
 * description: Connections from file "cities.txt" are loaded into a graph
 * which is represented as 2D matrix. Then connections are disrupted by
 * the user.
 * Afterwards, connections can be searched in the graph using the Dijkstra's
 * shortest path algorithm. If connection is found, the cities on the way
 * are printed in order in which they were visited. Then the time needed is
 * also printed. If if is not possible to reach the final destination,
 * "UNREACHABLE" is printed.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include "heap.h"

// The number of cities on the map
#define SIZE 12

//returns name of the city of a given number
char *cityName(int n) {
  switch (n) {
    case 0:
      return "Leeuwarden";
    case 1:
      return "Groningen";
    case 2:
      return "Meppel";
    case 3:
      return "Zwolle";
    case 4:
      return "Enschede";
    case 5:
      return "Amsterdam";
    case 6:
      return "Den Helder";
    case 7:
      return "Den Haag";
    case 8:
      return "Utrecht";
    case 9:
      return "Eindhoven";
    case 10:
      return "Nijmegen";
    case 11:
      return "Maastricht";
    default:
      return "";
  }
}
```

```
58
59  // returns code number for the city
60  int cityNumber(char *str) {
61    if (!strcmp(str, "Leeuwarden")) {
62      return 0;
63    }
64    if (!strcmp(str, "Groningen")) {
65      return 1;
66    }
67    if (!strcmp(str, "Meppel")) {
68      return 2;
69    }
70    if (!strcmp(str, "Zwolle")) {
71      return 3;
72    }
73    if (!strcmp(str, "Enschede")) {
74      return 4;
75    }
76    if (!strcmp(str, "Amsterdam")) {
77      return 5;
78    }
79    if (!strcmp(str, "Helder")) {
80      return 6;
81    }
82    if (!strcmp(str, "Haag")) {
83      return 7;
84    }
85    if (!strcmp(str, "Utrecht")) {
86      return 8;
87    }
88    if (!strcmp(str, "Eindhoven")) {
89      return 9;
90    }
91    if (!strcmp(str, "Nijmegen")) {
92      return 10;
93    }
94    if (!strcmp(str, "Maastricht")) {
95      return 11;
96    }
97    printf("INVALID CITY NAME\n");
98    abort();
99  }
100
101 // prints the cities in order in which they are visited on the way
102 void printReverse(int prev[SIZE], int current, int s) {
103   if (current != s) {
104     printReverse(prev, prev[current], s);
105   }
106   printf("%s\n", cityName(current));
107 }
108
109 // the Dijskra's shortest path algorithm
110 void dijkstraSearch(const int conn[SIZE][SIZE], const int s, const int e) {
111   int weight[SIZE]; //the time of going from S to current
112   int prev[SIZE]; // we got to node a from the value of prev[a]
113   int visited[SIZE] = {0}; // keep track of which city we visited
114   int currentCity = s; // starting position
115   for (int i = 0; i < SIZE; i++) {
116     weight[i] = INT_MAX / 2;
117   }
118   weight[s] = 0;
```

```
119
120   Heap todo = makeHeap();
121   enqueue(weight[s], &todo, s);
122
123   while (!isEmptyHeap(todo) && currentCity != e) {
124
125   while(visited[currentCity] == 1 && !isEmptyHeap(todo)){ // avoiding
          duplicates in the heap
126     currentCity = removeMin(&todo);
127   }
128     visited[currentCity] = 1;
129
130     for (int i = 0; i < SIZE; i++) { //assign new weight of connection to i
131       if (visited[i] == 0 && conn[currentCity][i] != 0 &&
132           weight[i] > weight[currentCity] + conn[currentCity][i]) { //new
              connection is shorter
133         weight[i] = weight[currentCity] + conn[currentCity][i]; // new, best
              time of connection
134         prev[i] = currentCity; // direction of the connection
135         enqueue(weight[i], &todo, i); // add to heap the new weight (time)
              and the city number
136       }
137     }
138   }
139
140   if (currentCity != e) { // if we do not terminate on the target city it is
         unreachable
141     printf("UNREACHABLE\n");
142   } else {
143     printReverse(prev, currentCity, s); // print the stations we went
          through
144     printf("%i\n", weight[currentCity]); // print the time needed to get
          there
145   }
146   freeHeap(todo);
147 }
148
149 int main(void) {
150   // input connection values into the graph
151   FILE *in_file = fopen("cities.txt", "r"); // read file with cities
152   int pointA, pointB, distance;
153   int conn[SIZE][SIZE]; //array of connections
154   memset(conn, 0, sizeof(int) * SIZE * SIZE); // set all values to 0
155   fscanf(in_file, "%i %i %i", &pointA, &pointB, &distance); // scan values
          from the file
156   while (pointA != -1) { // input pointA, pointB and distance into the graph
157     conn[pointA][pointB] = distance;
158     fscanf(in_file, "%i %i %i", &pointA, &pointB, &distance);
159   }
160
161   // input of disruptions
162   int disruptions;
163   char city1[20], city2[20];
164   scanf("%i", &disruptions);
165   for (int i = 0; i < disruptions; i++) {
166     scanf("%s", city1);
167     if (!strcmp(city1, "Den")) { // some cities start with 'Den_'
168       scanf("%s", city1);
169     }
170     scanf("%s", city2);
171     if (!strcmp(city2, "Den")) {
```

```
      scanf("%s", city2);
    }
    // cancel the connection between the cities
    conn[cityNumber(city1)][cityNumber(city2)] = 0;
    conn[cityNumber(city2)][cityNumber(city1)] = 0;
  }

  // input of destinations
  scanf("%s", city1);
  if (!strcmp(city1, "Den")) {
    scanf("%s", city1);
  }
  while (strcmp(city1, "!") != 0) {
    scanf("%s", city2);
    if (!strcmp(city2, "Den")) {
      scanf("%s", city2);
    }
    dijkstraSearch(conn, cityNumber(city1), cityNumber(city2));
    scanf("%s", city1);
    if (!strcmp(city1, "Den")) {
      scanf("%s", city1);
    }
  }
  fclose(in_file);
  return 0;
}
```

Listing 2: heap.c

```
/* name: heap.c
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 23.03.2022
 * description: a structure of heap. The root is has a minimal value. The
    heap
 * operates on connection times, but it returns the number of a city with
 * the smallest value.
 */

#include "heap.h"

void doubleHeapSize(Heap *hp) {
  // creating arrays of double size
  int *newArr = malloc((hp->size) * 2 * sizeof(int));
  int *newCityArr = malloc((hp->size) * 2 * sizeof(int));
  // copying the content to the doubled arrrays
  for (int i = 0; i < hp->size; i++) {
    newArr[i] = hp->array[i];
    newCityArr[i] = hp->cityNumber[i];
  }
  hp->size *= 2; // double parameter
  free(hp->array);
  free(hp->cityNumber);
  hp->array = newArr; // assign the new arrays
  hp->cityNumber = newCityArr;
}

void upheap(Heap *hp, int n) {
  int parent = n / 2;
```

```
32    if (n > 1 && hp->array[n] < hp->array[parent]) { // if upheap is needed
          swap corresponding values
33      swap(&(hp->array[n]), &(hp->array[parent]));
34      swap(&(hp->cityNumber[n]), &(hp->cityNumber[parent]));
35      upheap(hp, parent);
36    }
37  }
38
39  void downheap(Heap *hp, int n) {
40    int indexMax = n; // the node (position) we are downheaping
41    if (hp->front < 2 * n + 1) {
42      return;
43    }
44    if (hp->array[n] > hp->array[2 * n]) {
45      indexMax = 2 * n;
46    }
47    if (hp->front > 2 * n + 1 && hp->array[indexMax] > hp->array[2 * n + 1]) {
48      indexMax = 2 * n + 1;
49    }
50    if (indexMax != n) { // we have not passed any conditions, we are at the
          right node
51      swap(&(hp->array[n]), &(hp->array[indexMax]));
52      swap(&(hp->cityNumber[n]), &(hp->cityNumber[indexMax]));
53      downheap(hp, indexMax); // propagate further
54    }
55  }
56
57  // initialising a heap
58  Heap makeHeap() {
59    Heap h;
60    h.array = malloc(1 * sizeof(int));
61    assert(h.array != NULL);
62    h.cityNumber = malloc(1 * sizeof(int));
63    assert(h.cityNumber != NULL);
64    h.front = 1;
65    h.size = 1;
66    return h;
67  }
68
69  int isEmptyHeap(Heap h) {
70    return (h.front == 1);
71  }
72
73  void heapEmptyError() {
74    printf("heap empty\n");
75    abort();
76  }
77
78  // add new element to the heap
79  void enqueue(int n, Heap *hp, int city) {
80    int fr = hp->front;
81    if (fr == hp->size) {
82      doubleHeapSize(hp);
83    }
84    hp->array[fr] = n;
85    hp->cityNumber[fr] = city;
86    upheap(hp, fr);
87    hp->front = fr + 1;
88  }
89
90  // removes the smallest (top) value from the heap
```

```
int removeMin(Heap *hp) {
  int n;
  if (isEmptyHeap(*hp)) {
    heapEmptyError();
  }
  n = hp->cityNumber[1];
  hp->front--;
  hp->array[1] = hp->array[hp->front];
  hp->cityNumber[1] = hp->cityNumber[hp->front];
  downheap(hp, 1);
  return n;
}

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void freeHeap(Heap hp) {
  free(hp.array);
  free(hp.cityNumber);
}

void printHeap(Heap hp) {
  for (int idx = 1; idx < hp.front; ++idx) {
    printf("%d ", hp.array[idx]);
  }
  printf("\n");
  for (int idx = 1; idx < hp.front; ++idx) {
    printf("%d ", hp.cityNumber[idx]);
  }
  printf("\n");
}
```

Listing 3: heap.h

```
/* name: heap.h
 *
 * author: Michal Tesnar (m.tesnar@student.rug.nl, s4740556)
 * author: Wojciech Trejter (w.trejter@student.rug.nl, s4296850)
 *
 * date: 23.03.2022
 * description: A header for a heap. It contains two arrays; one with
    weights, which
 * the heap is sorted by. The other one are corresponding city numbers.
 */

#ifndef HEAP_H
#define HEAP_H

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <assert.h>

typedef struct Heap {
  int *array; // weights of connections
  int *cityNumber; // identifiers of cities
  int front;
  int size;
```

```
24 } Heap;
25
26 void doubleHeapSize (Heap *hp);
27 int removeMin(Heap *hp);
28 void enqueue(int n, Heap *hp, int city);
29 void heapEmptyError();
30 int isEmptyHeap(Heap h);
31 Heap makeHeap();
32 void upheap(Heap *hp, int n);
33 void downheap(Heap *hp, int n);
34 void swap(int *a, int *b);
35 void freeHeap(Heap hp);
36 void printHeap(Heap hp);
37 void heapSort(int n, int arr[]);
38
39 #endif
```

Listing 4: cities.txt

```
0 1 34
0 2 40
1 0 34
1 2 49
2 3 15
2 0 40
2 1 49
3 2 15
3 4 50
3 8 51
3 10 77
4 3 50
5 8 26
5 7 46
5 6 77
6 5 77
7 5 46
7 9 89
8 5 26
8 9 47
8 3 51
9 8 47
9 10 55
9 11 63
9 7 89
10 9 55
10 3 77
10 11 111
11 9 63
11 10 111
-1 -1 -1

* numbers assigned to cities *
0 Leeuwarden
1 Groningen
2 Meppel
3 Zwolle
4 Enschede
5 Amsterdam
6 Den Helder
7 Den Haag
8 Utrecht
```

```
9 Eindhoven
10 Nijmegen
11 Maastricht

* weights of all connections *
Amsterdam Den Haag 46   -> 5 7 46 || 7 5 46
Amsterdam Den Helder 77 -> 5 6 77 || 6 5 77
Amsterdam Utrecht 26    -> 5 8 26 || 8 5 26
Den Haag Eindhoven 89   -> 7 9 89 || 9 7 89
Eindhoven Maastricht 63 -> 9 11 63 || 11 9 63
Eindhoven Nijmegen 55   -> 9 10 55 || 10 9 55
Eindhoven Utrecht 47    -> 9 8 47 || 8 9 47
Enschede Zwolle 50      -> 4 3 50 || 3 4 50
Groningen Leeuwarden 34 -> 1 0 34 || 0 1 34
Groningen Meppel 49     -> 1 2 49 || 2 1 49
Leeuwarden Meppel 40    -> 0 2 40 || 2 0 40
Maastricht Nijmegen 111 -> 11 10 111 || 10 11 111
Meppel Zwolle 15        -> 2 3 15 || 3 2 15
Nijmegen Zwolle 77      -> 10 3 77 || 3 10 77
Utrecht Zwolle 51       -> 8 3 51 || 3 8 51
```