Assignment 5: Trains - An Adventure Drama
Programming report

s5524040 & s5622360

Algorithms and Data Structures 2023-2024

1 Problem description

General input: The input consists of a finite number of train networks and their stations, as well as possible disruptions. Also, each train network has a list of queries/requests to get from one point on the "map" to another.

General output: For each query, the program should output the path from the starting point to the destination as well as the total travelled distance if the path exists and "UNREACHABLE" otherwise.

2 Problem analysis

To represent the map of the stations of a train network we will be using a graph, because it's a convenient way to represent stations as nodes and routes as edges, also, having a directed graph will allow us to apply the Dijkstra's algorithm to retrieve the shortest path between two stations.

The Dijkstra's algorithm can be described as following:

```
algorithm Dijkstra(Graph, source)
  for each vertex v in Graph:
    dist[v] = infinity
    [v] = undefined
  dist[source] = 0
  Q = the set of all nodes in Graph
  while Q is not empty:
    u = node in Q with smallest dist[ ]
    remove u from Q
    for each neighbor v of u:
      alt = dist[u] + dist_between(u, v)
      if alt ¡ dist[v]
        dist[v] = alt
        previous[v] = u
return previous[ ]
```

3 Program design

As suggested by the program description, the program first reads the number of networks, and then "jumps" to a while loop where the guard is the number of networks greater than 0. To ensure a correct execution of the loop, we decrement the number of networks and this way we're defining a block of code which will execute on every single network. Thus, after getting the information related to the train network and it's number of stations, we are building an initial graph which represents the train network, using an auxiliary function (makeGraph), and adding the stations to the graph using the function newGraphNode. By having a graph which represents the stations map, we will be able to apply the Dijkstra's algorithm for shortest path later on. It is worth noting that Dijkstra's algorithm was implemented using a min-priority queue, which also has a heap as its structure.
The attempted implementation of the A* algorithm follows a similar mentality, but the heuristic values are achieved using the mathematical formulas for the distance between two points in a 2D plane.

Design choice.
We define the functions makeGraph, newGraphNode, addEdge and removeEdge to create and manipulate the graph which represents the train stations map. Also, we are defining the structure "adjListNode" which holds information about the current station and a pointer to the next one and the structure "graphStructure" which holds the number of the vertices in the graph and a list of nodes (stations).
An important thing to note, when implementing Dijkstra's algorithm, the value of INFINITY appears. Since that is not easily available in a program, we are substituting it with the maximum value that an integer variable can hold.

Time complexity. The time complexity of the shortest path retrieval algorithm (Dijkstra) is O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.

4 Evaluation of the program

We test the program with the following input:
3, 2, 0 A, 1 B, 1, 0 1 1, 1, A, B, A, B, !, 4, 0 Amsterdam, 1 Berlin, 2 Paris, 3 Marseille, 5, 0 1 382, 0 2 238, 1 2 501, 1 3 709, 2 3 182, 1, Berlin, Marseille, Berlin, Marseille, Amsterdam, Marseille, !, 4, 0 Gallivare, 1 Ostersund, 2 Mora, 3 Trondheim, 3, 0 1 803, 1 2 309, 1 3 1609, 0, Mora, Gallivare, Trondheim, Mora, !

And get the expected output:
UNREACHABLE, Berlin, Paris, Marseille, 683, Amsterdam, Paris, Marseille, 420, Mora, Ostersund, Gallivare, 1112, Trondheim, Ostersund, Mora, 1918

On top of the correct output, it is also important to ensure correct memory management, which can become a tedious task with Data Structures. It is essential that only the required amount of memory is allocated and that every allocation is freed up in the end. For these purposes, we've used Valgrind to test our program and make sure that no leaks are possible:

==2589==
==2589== HEAP SUMMARY:
==2589== in use at exit: 0 bytes in 0 blocks
==2589== total heap usage: 137 allocs, 137 frees, 7,971 bytes allocated
==2589==
==2589== All heap blocks were freed – no leaks are possible
==2589==
==2589== For lists of detected and suppressed errors, rerun with: -s
==2589== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

5 Process description

The program consists of accepting a number of networks as input, followed by data about each network : number of stations, the ID and name for each station, the connections, the disruptions between stations and queries that ask the for the path and distance between two stations. The program stops accepting queries upon encountering "!" as input.
Designing the data structures used in the program was not the hard part, but rather implementing the min-priority queue through a heap was difficult in the beginning. At first, the correct implementation eluded us and lead to us spending a lot of time running around in circles trying to figure out a solution. Upon understanding how to go about implementing, the process was much smoother and lead to the solution presented.
Another difficulty found in the program design was the memory deallocation. Initially, the process of deallocating memory seemed simple, but due to some design choices, a small leak always occurred. This lead to changing the approach to how to station names are stored and to the printing of the path.
Currently, however, the program presents an issue regarding Test case 6. We believe that it occurs due to using more memory than is available. This, in theory, could've been solved by changing the approach to using the 'pos[]' array of integers from the heap struct, however we have not reached a good solution without it so we gave up on that idea.
By developing this program we became more accustomed to using Dijkstra's algorithm and correctly implementing more complex data structures.

6 Conclusions

The nature of the program has determined us to use graphs to represent the map of the train stations. The graph can be easily manipulated to add or remove stations and routes. Although processing the input and building the corresponding graph is a bit tedious and time consuming in the beginning, it surely pays off in the end later on because Dijkstra's algorithm can be applied on directed graphs to retrieve the shortest path while also being very effective in terms of runtime and memory consumption

7 Appendix: program text

Listing 1: Input processing

```c
#include <stdio.h>
#include <stdlib.h>
#include "priorityQueue.h"
#include <string.h>

int main(int argc, char** argv) {
  //First we read the number of networks to expect from the input
  int networks;
  scanf("%d", &networks);
  while(networks) {
    networks--;
    //Now we start reading the input for each network
    int stationsNr;
    char bufferCleaner;
    scanf("%d", &stationsNr);
    //This list will hold all the stations and the data relevant to them
    adjListNode* stations = malloc(sizeof(adjListElement)*(stationsNr));
    //Here we make the graph used to apply Dijkstra's algorithm
    graph stationsMap = makeGraph(stationsNr);
    int cStationsNr = stationsNr;
    while(cStationsNr){
      cStationsNr--;
      //This accepts the input for each station
      int idNum;
      scanf("%d ", &idNum);
      char* staionNameInput = inputName();
      stations[idNum] = newGraphNode(idNum, 0);
      stations[idNum]->stationName = malloc((strlen(staionNameInput)+1)*sizeof(char));
      strcpy(stations[idNum]->stationName, staionNameInput);
      free(staionNameInput);
    }
    //This section handles the connections
    int connections;
    scanf("%d\n", &connections);
    int i = 0;
    while(i<connections) {
      i++;
      int place1, place2, distance;
      //each connection is added as an edge
      scanf("%d %d %d", &place1, &place2, &distance);
      addEdge(stationsMap, stations[place1], stations[place2], distance);
    }
    //This section handles the disruptions
    int disruptions;
    scanf("%d", &disruptions);
    //The buffer cleaner is here to handle the appearence of a NULL character
    bufferCleaner = getchar();
```

```
      if(bufferCleaner != '\n') {
        printf("detected issue");
      }


    char* toremove;
    while(disruptions>0) {
      //This accepts a name as an input and removes the edge from the graph, it will ignore incorrect disruptions
      toremove = inputName();
      int stationToRemove1 = findInList(stations, toremove, stationsNr);
      free(toremove);
      toremove = inputName();
      int stationToRemove2 = findInList(stations, toremove, stationsNr);
      removeEdge(stationsMap, stations[stationToRemove1], stations[stationToRemove2], &stationsNr);
      disruptions--;
      free(toremove);
    }
    //This section handles the queries
    int queryID1, queryID2;
    char* queryInput = inputName();
    while(queryInput[0]!= '!') {
      //The queries consist of two names and the path and distance between them or UNREACHABLE if there is no path between them
      int reached = 0;
      queryID1 = findInList(stations, queryInput, stationsNr);
      free(queryInput);
      queryInput = inputName();
      queryID2 = findInList(stations, queryInput, stationsNr);
      dijkstraShortestPath(stationsMap, queryID1, queryID2, &reached, stations);
      if(reached==0) {
        printf("UNREACHABLE\n");
      }
      free(queryInput);
      queryInput = inputName();
    }
    //Now we free all the data structures we used
    free(queryInput);
    freeGraph(stationsMap);
    freeNodeList(stations, stationsNr);
  }
}
```

Listing 2: Graphs manipulation

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


//This function is similar to the one provided in the code for the 4th assignment and its purpose is to accept
//a name as input
char *inputName() {
  int strLen = 100;
  int length = 0;

  char character = getchar();
```

```c
  char *name  = malloc(sizeof(char)*(strLen+1));

  while(character!='\n') {
    name[length] = character;
    length++;
    if(length >= strLen) {
      strLen = 2*strLen;
      name = realloc(name, (strLen+1)*sizeof(char));
    }
    character = getchar();
  }
  name[length] = '\0';
  return name;
}


//This function makes and returns a new graph node using
adjListNode newGraphNode (int id, int distance) {
  adjListNode new = malloc(sizeof(adjListElement));
  new->stationIDNumber = id;
  new->stationDistance = distance;
  new->next = NULL;
  new->distance = distance;
  return new;
}

//This function creates and empty graph with the number of nodes equal to
    the one inputed
graph makeGraph (int nr) {
  graph newGraph = malloc(sizeof(graph)*(nr+1));
  newGraph->vertices = nr;
  newGraph->graphArray = malloc(sizeof(adjListElement)*(nr+1));
  for(int i = 0; i < nr; i++) {
    newGraph->graphArray[i].head = NULL;
  }
  return newGraph;
}

//This function adds an edge to the graph's adjacency list using the data
    provided
void addEdge(graph elGrapho, adjListNode source, adjListNode destination,
    int distance) {
  adjListNode new1 = newGraphNode(destination->stationIDNumber, distance);
  new1->next = elGrapho->graphArray[source->stationIDNumber].head;
  elGrapho->graphArray[source->stationIDNumber].head = new1;

  //The graph is undirected so it must go from vertex A to B and from vertex
      B to A
  adjListNode new2 = newGraphNode(source->stationIDNumber, distance);
  new2->next = elGrapho->graphArray[destination->stationIDNumber].head;
  elGrapho->graphArray[destination->stationIDNumber].head = new2;
}



//This function removes and edge from the graph's adjacency list, ignoring
    the invalid deletions
void removeEdge (graph elGrapho, adjListNode toRemove1, adjListNode
    toRemove2, int *totalStations) {
  //Both the edge from A to B and from B to A must be removed, since the
      graph is undirected
```

```
  adjListNode currentNode = elGrapho->graphArray[toRemove1->stationIDNumber].head;
  adjListNode previousNode = NULL;
  //This loop looks for the other vertex incident with the edge to be removed
  while(currentNode !=NULL && currentNode->stationIDNumber!=toRemove2->stationIDNumber) {
    previousNode = currentNode;
    currentNode = currentNode->next;
  }


  //Upon finding it,if it's found, it's removed by freeing the memory
  if(currentNode!=NULL) {
    if(previousNode == NULL) {
      elGrapho->graphArray[toRemove1->stationIDNumber].head = currentNode->next;
    } else {
      previousNode->next = currentNode->next;
    }
    free(currentNode);
  }

  //The same procedure is applied for the second vertex of the node
  currentNode = elGrapho->graphArray[toRemove2->stationIDNumber].head;
  previousNode = NULL;
  while(currentNode !=NULL && currentNode->stationIDNumber!=toRemove1->stationIDNumber) {
    previousNode = currentNode;
    currentNode = currentNode->next;
  }

  if(currentNode != NULL) {
    if(previousNode == NULL) {
      elGrapho->graphArray[toRemove2->stationIDNumber].head = currentNode->next;
    } else {
      previousNode->next = currentNode->next;
    }
    free(currentNode);
  }

  return;
}

//This funtion looks into the list of nodes and returns the position of the station in the list
int findInList(adjListNode* stations, char* name, int stationsNr) {
  for(int i = 0; i<stationsNr; i++) {
    if(strcmp((*stations[i]).stationName, name)==0) {
      return i;
    }
  }
  return -1;
}

//This function frees the graph
void freeGraph(graph g) {
  for(int i = 0; i<g->vertices; i++) {
    adjListNode newNode = g->graphArray[i].head;
    adjListNode next;
```

```c
    while(newNode!=NULL) {
      next = newNode->next;
      freeNode(newNode);
      newNode = next;
    }
  }
  free(g->graphArray);
  free(g);
}

//This funtcion frees a particular node
void freeNode(adjListNode node) {
  free(node);
}

//This function frees the adjacency list
void freeNodeList(adjListNode* list, int number) {
  for(int i=0; i< number; i++) {
    free(list[i]->stationName);
    freeNode(list[i]);
  }
  free(list);
}
```

Listing 2: Path priority

```c
#include "priorityQueue.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

//This fucntion frees the memorya allocated to a heap node
void freeHeapNode(heapNode node) {
  free(node);
}

//This fucntion free the memory allocated to a heap
void freeHeap(heap h, int source) {

  free(h->array);
  free(h->pos);
  free(h);
}

//This structure is used to remember the path discovered in the Dijkstra's
    algorithm
typedef struct pathHolder{
  int* IDs;
  int distance;
}pathHolder;


//This fucntion creates and allocates memory to a new node for the heap
heapNode newHeapNode(int distance, int v, adjListNode node) {
  heapNode new = malloc(sizeof(heapElem));
  new->dist = distance;
  new->v = v;
  new->node = node;
  return new;
}

```

```c

//This function creates and allocates memory for a heap
heap makeHeap(int capacity) {
  heap new = malloc(sizeof(heapStructure));
  new->pos = malloc(capacity*sizeof(int));
  new->size = 0;
  new->capacity = capacity;
  new->front  = 0;
  new->array = malloc(capacity*sizeof(heapNode));
  for(int i = 0; i < capacity; i++) {
    new->array[i] = NULL;
  }
  return new;
}

//Thsi fucntion swaps two nodes of a heap
void swapHeapNodes(heapNode* a, heapNode* b) {
  heapNode temp = *a;
  *a = *b;
  *b = temp;
  return;
}

//This function fixes the heap from the given id in order for it to be a minimum Heap
//It also updates the position of nodes when they are swapped
void upHeap(heap h, int id) {
  int smallest, leftChild, rightChild;
  smallest = id;
  leftChild = id*2+1;
  rightChild = id*2+2;

  if(leftChild < h->size && h->array[leftChild]->dist < h->array[smallest]->dist) {
    smallest = leftChild;
  }

  if(rightChild < h->size && h->array[rightChild]->dist < h->array[smallest]->dist) {
    smallest = rightChild;
  }

  if(smallest != id) {
    heapNode smallestNode = h->array[smallest];
    heapNode idNode = h->array[id];
    h->pos[smallestNode->v] = id;
    h->pos[idNode->v] = smallest;


    swapHeapNodes(&h->array[smallest], &h->array[id]);

    upHeap(h, smallest);
  }
  return;
}


//This function returns the minimum of a heap and removes it from said structure
heapNode getMinimum(heap h) {
  if(isEpmtyHeap(h)) {
```

```c
    return NULL;
  }

  heapNode root = h->array[0];
  heapNode lastNode = h->array[h->size - 1];
  h->array[0] = lastNode;

  h->pos[root->v] = h->size - 1;
  h->pos[lastNode->v] = 0;


  h->size = h->size-1;

  upHeap(h, 0);

  return root;
}

//This fucntion modifies the distance stored at vertex v/the priority stored
    at vertex v
void decreaseDistance(heap h, int vertex, int distance) {
  int i = h->pos[vertex];
  h->array[i]->dist = distance;

  //Travels down the heap until it is correct
  while(i && h->array[i]->dist < h->array[(i-1)/2]->dist) {
    h->pos[h->array[i]->v] = (i-1)/2;
    h->pos[h->array[(i-1)/2]->v] = i;
    swapHeapNodes(&h->array[i], &h->array[(i-1)/2]);
    i = (i-1)/2;
  }
}

//This checks whether the heap is empty
int isEpmtyHeap(heap h) {
  return h->size == 0;
}


//This checks whether an element is still in the heap
int isInHeap(heap h, int v) {
  if(h->pos[v] < h->size) {
    return 1;
  }

  return 0;
}


//This fucntion is used to print a string
void printString(char *str) {
  for(int i = 0; str[i] != '\0'; i++) {
    printf("%c", str[i]);
  }
  printf("\n");
}

//This is the implementation of Dijkstra's algorithm using and adjacency
   list and a min-priority queue represented by a heap
void dijkstraShortestPath(graph g, int source, int destination, int *
   reacheable, adjListNode* stations) {
```

```c
  pathHolder* p = malloc(g->vertices*sizeof(pathHolder));
  int vertices = g->vertices;
  int* prev = malloc((vertices+1)*sizeof(int));
  //This si the empty priority queue
  heap S = makeHeap(vertices);

  //This loop declares the array which will be used to print the path later
  for(int i = 0; i < vertices; i++) {
    p[i].IDs = malloc((vertices+1)*sizeof(int));
  }

  //This is the initialization of the dist[source] = 0 and the other source-related values
  //newHeapNode is going to be used as the add_with_priority() function
  p[source].distance = 0;
  prev[source] = -1;
  S->array[source] = newHeapNode(p[source].distance, source, g->graphArray[source].head);
  S->pos[source] = source;

  //This initiates the other values different from the source
  //intead of INFINITY, the maximum value that can be held by an integer variable is used
  for(int i = 0; i < vertices; i++) {
    if(i != source) {
      p[i].distance = INT_MAX;
      prev[i] = -1;
      S->array[i] = newHeapNode(p[i].distance, i, g->graphArray[i].head);
      S->pos[i] = i;
    }

  }


  decreaseDistance(S, source, p[source].distance);
  S->size = vertices;
  //Here starts the loop to find the path
  //while S is not empty
  while(!isEpmtyHeap(S)) {
    //u <- S. extract_min()
    heapNode minimumHeapNode = getMinimum(S);
    int u = minimumHeapNode->v;
    //This is used to go through the incident nodes
    adjListNode crawler = g->graphArray[u].head;
    while(crawler != NULL) {
      //v is the node adjacent to the minimum node/u
      int v = crawler->stationIDNumber;
      //this holds the current distance from the source
      int alt = p[u].distance + crawler->distance;
      if(isInHeap(S, v) && p[u].distance!=INT_MAX && alt< p[v].distance) {
        *reacheable = 1;
        //This modifies the distance from the source to the node v
        p[v].distance = alt;
        prev[v] = u;
        //this reduces priority/ S.decrease_priority(v, alt), distance is the equivalent to priority
        decreaseDistance(S, v, alt);
      }
      crawler = crawler->next;

    }
```

```
            //After using the node, we free it to avoid memeory leaks
            free(minimumHeapNode);
        }

        //Here the printing is done, if the point is reacheable
        int nr =0;
        if(*reacheable!=0) {
            if(p[destination].distance == INT_MAX) {
                *reacheable = 0;
                return;
            }

            int j = destination;

            while(j!=-1) {
                p[destination].IDs[nr] = j;
                nr++;
                j = prev[j];
            }
            for(int z = nr-1; z>=0;z--) {
                printf("%s\n", stations[p[destination].IDs[z]]->stationName);

            }
            printf("%d\n", p[destination].distance);
        }

        //Now we free the used structures
        free(prev);
        for(int i = 0; i < vertices; i++) {
            free(p[i].IDs);
        }
        free(p);
        freeHeap(S, source);

        return;
    }
```