Assignment 5: "Maze"
Programming Report

s5741300 and s5951178
Algorithms and Data Structures in C (2024-2025)

1. Problem description

General:
Write a C program that finds the shortest path through the *Inverto-Maze*. The maze is modelled as a directed graph where each chamber (vertex) is connected by tunnels (edges) that initially allow travel in a given direction. Some chambers are equipped with the `R`-button that, when pressed, reverses the direction of all tunnels. The program must determine the shortest route from the entrance to the exit and its length, and indicate on the path the chambers where the button was pressed.

Input-output behavior:
The input begins with two integers, n (number of chambers) and m (number of tunnels). The next line lists all chambers that have an `R`-button, terminated by `-1`. This is followed by m lines, each containing three integers: a, b, and ℓ, indicating that there is a tunnel from chamber a to chamber b with length ℓ (the tunnel is initially traversable in the direction $a \to b$).
The output first displays a single integer: the length of the shortest path from entrance to exit. This is then followed by the sequence of chambers (one per line) along the shortest path. A button press is indicated by appending an "R" after the chamber number (except for the exit).

Example input:

```
8 11
6 3 4 1 -1
7 3 3
1 4 33
5 2 2
2 4 5
2 7 8
3 6 15
1 7 7
6 7 12
6 5 1
4 8 10
8 5 20
```

Example output:

```
42
1
```

```
7
3 R
7
2
5
6 R
5
2
4
8
```

2. Problem analysis

The challenge is a shortest-path problem with the possibility of reversing the tunnel directions by pressing a button. This effect is modelled by splitting each chamber into two *states*:

- → **State 0:** Normal maze configuration.
- → **State 1:** Reversed maze configuration.

Thus, each chamber is represented as two nodes, and there are $2 \times n$ states possible. The transition between states (by pressing the button) is allowed at zero cost when a chamber has a button. In addition, when traversing a tunnel, the state remains unchanged. Using this expanded graph model, Dijkstra's algorithm can be applied to compute the shortest path from the entrance to the exit (in either reversed or original state). Here it is clear that an *Analogy* approach was used, since the problem was merely adapted to a state where Dijkstra's algorithm could be easily applied.

During the relaxation step the algorithm considers two kinds of transitions:

- → **Traversing a tunnel:** Moving from chamber u to chamber v while keeping the current state.
- → **Pressing the `R`-button:** In a chamber with an `R`-button, the state can be toggled at zero cost.

In addition, to keep track of distances and be able to take the node with the smallest one, a priority queue was implemented through a reverse Heap. Properties of the heap were then used to retrieve smallest distance efficiently.

The implementation of the Dijkstra algorithm in pseudocode is the following:

algorithm DijkstraWithState(G, G', n, cButtons)
  input: Original graph G, reversed graph G', number of chambers n,
    array cButtons indicating which chambers have an R-button
  result: Shortest path from chamber 1 to chamber n, with button-presses marked
  for each chamber u = 1 to n **do**
    for s $\in \{0,1\}$ **do**
      distance[u,s] ←**if** u=0, s=0 **then** 0 **else** ∞ /* distance initialized */
      previous[u,s] ← -1 /* path links initialized */
      Insert node (u,s) into the min-heap H with key distance[u,s]
  while H is not empty **do**
    (u,s) ←node in H with minimal value of distance/* using PopPriorityQueue */
    if cButtons[u] is true **then**
      if distance[u,s] is less than distance[u,1-s] **then**
        distance[u,1-s] ←distance[u,s]/* No cost in pressing the button */

previous[u,1-s] ←(u,s)
Update node (u,1-s) in H/∗ Restore the Heap property ∗/
forall z ∈ H with (u, z) ∈ edges(G^*) **do**/∗ G^* active graph, depends on s ∗/
if distance[u,s] + weight(u,z) < distance[z,s] **then**
distance[z,s] ←distance[u,s] + weight(u,z)
previous[z,s] ←(u,s)
Update node (z,s) in H

3. Program design

The solution is implemented in two main modules, `graph.c/graph.h` and `maze.c`. The important choices for translating the analytical approach to code were as follows:

1. **Data Structures:**

   → **Graph Representation:** Two separate neighbor lists are built: one for the original tunnel directions and one for the reversed configuration. The Graphs are stored as arrays of linked list, where entry *i* of the array represents all neighbors of node *i + 1*. This is because nodes are stored in base 0 representation. All these functionalities and implementations are included in the `graph.c/graph.h` module.

   → **Min-Heap:** A priority queue (min-heap) is used for Dijkstra's algorithm. Each heap node stores a chamber, its current distance from the entrance, and its state. Additionally, the priority queue holds its current size and a positional array, that at entry *i* shows where node *i + 1* is currently in the heap. This allows for more efficient look-ups during Dijkstra's algorithm. These implementations and the main functionality of the program is included in `maze.c`

2. **Initialization:**

   → Allocate an array of size $2n$ for distances (which allows for more intuitive and efficient lookups and prevents accessing deleted memory from the Heap), and initialize all distances to `INT_MAX` (representation of ∞) except for the entrance (at state 0) which is set to 0.

   → Initialize the min-heap with all $2n$ nodes and distances as chosen above.

3. **Dijkstra's Algorithm:**

   → The algorithm repeatedly extracts the node with the smallest distance, by popping the top element from the heap, replacing it with the last element and restoring heap properties.

   → For each extracted node, two kinds of relaxations are performed:

     → *Button press:* If the chamber has a button, relax the edge from the current state to the opposite state with zero cost.

     → *Tunnel traversal:* Traverse each outgoing tunnel in the active graph (depending on the current state) and relax the distance to the neighboring node.

4. **Path reconstruction:**

   → After the algorithm terminates, the shortest path is reconstructed by backtracking from the exit node using a `previous` array.

→ When the state changes between consecutive nodes in the path, an R is appended to indicate a button press.

Time complexity considerations

→ **Building the adjacency list:** Reading the m tunnels and creating n adjacency lists takes $O(m)$ time overall, as each edge is simply appended to a linked list.

→ **Modified Dijkstra's algorithm:** Each chamber has two states, leading to $2n$ list nodes in the priority queue. If m is the total number of edges in the original graph, then the reversed graph also has m edges, so we have $2m$ edges in our graph. Using a binary heap, each extract-min operation takes $O(\log(2n)) \equiv O(\log n)$ time. So with $2n$ nodes and $2m$ edges, the running time is:

$$O((n + m)\log n).$$

→ **Heap operations (heapifyUp and heapifyDown):** Each insertion, removal, or distance decrease triggers heapifyUp or heapifyDown. Both are $O(\log n)$ in the worst case.

4. Evaluation of the program

The program was successfully accepted by Themis. All possible test cases we could come up with produced the right output. This includes the given example test case, as well all the Themis test cases like:

```
17 20
1 7 14 10 -1
2 1 11
2 11 4
3 2 8
4 3 1
5 4 7
6 5 5
7 6 23
7 8 2
9 8 9
9 17 0
11 12 7
12 13 4
13 14 5
14 15 2
15 14 10
15 16 3
16 10 10
16 15 1
10 16 3
10 9 0
```

Which indeed produced the desired output:

```
133
1 R
```

```
2
3
4
5
6
7 R
6
5
4
3
2
11
12
13
14 R
15
16
10 R
9
17
```

Additionally, we apply `valgrind` to check for possible memory leaks. With the above input, we get the following summary:

```
==1656==
==1656== HEAP SUMMARY:
==1656==     in use at exit: 0 bytes in 0 blocks
==1656==   total heap usage: 85 allocs, 85 frees, 4,276 bytes allocated
==1656==
==1656== All heap blocks were freed -- no leaks are possible
==1656==
==1656== For lists of detected and suppressed errors, rerun with: -s
==1656== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

So no memory leaks or withstanding errors were found.

5. Process description

The problem took overall more than expected. Upon starting, we realized we were not really familiar with how to port graphs into C code, and that already took a lot of time. Eventually, after dealing with that implementation, we ran into the second wall: implementing priority queues with heaps. Not only that, but the heap had to be in reverse to how it was seen during the lectures. The final challenge was adapting Dijkstra's algorithm to the *Inverto-maze*. Nonetheless, once we devised the idea of keeping track of the current state and switching between reverse and original graph, all fell into place.

A remarkable error that we spent quite some time trying to fix was an out-of-bounds access to the Min Heap within Dijkstra's algorithm. We ultimately determined that using an extra array for keeping track of distances would not be such a bad idea after all, since ultimately all nodes were popped from the Heap and there was no feasible way to keep track of distances within it.

As in all previous assignments, we worked equally during the construction of the code. We took turns to code and, whilst one was typing, the other was thinking about how to implement the next few steps. In that sense, the work was split equally. From this assignment, we take away the efficiency of heaps when storing and keeping track of numbers in an ordinal sense, as well as complete understanding behind the cleverness of Dijkstra's algorithm. Additionally, having to think of representing a second maze via another graph in order to keep track of the *invertos* was certainly an outside-the-box approach that was really interesting to implement.

6. Conclusions

The implemented program successfully finds the shortest path through the maze and handles correctly the reversal of tunnel directions when buttons are pressed. The modified Dijkstra's algorithm with a state space provides efficiency. However, while efficient enough, Dijkstra's implementation is not the optimal for this problem. Since the problem asks about the path of one specific node, the optimal implementation of the solution should use an algorithm such as the A* algorithm for a performance gain.

7. Appendix: program text

graph.h

```c
#ifndef GRAPH_H
#define GRAPH_H

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

// Neighbours of a vertex are kept as linked lists.
typedef struct ListNode
{
    int vertex;
    int weight;
    struct ListNode *next;
} ListNode;

typedef ListNode *ListPointer;

ListPointer createNode(int nodeIndex, int weight);

void addEdge(ListPointer *neighbourList, int src, int dest, int
    weight);

void printGraph(ListPointer *neighbourList, int nodes);

#endif
```

graph.c

```c
#include <stdio.h>
```

```c
#include <stdlib.h>
#include "graph.h"

// Initialize a node that represents a neighbour.
ListPointer createNode(int nodeIndex, int weight)
{
    ListPointer newNode = (ListPointer)malloc(sizeof(ListNode));
    newNode->vertex = nodeIndex;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// Adds a node to the graph neighbourList, where each entry is a
    linked list.
void addEdge(ListPointer *neighbourList, int src, int dest, int
    weight)
{
    // Create a new node for dest.
    ListPointer newNode = createNode(dest, weight);

    // Insert at current head of list.
    newNode->next = neighbourList[src];
    neighbourList[src] = newNode;
}
```

maze.c

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

// Node in dijkstra's priority queue.
typedef struct HeapNode
{
    int vertex;
    int distance;
    int state; // Whether the node is in the reversed maze.
} HeapNode;

typedef struct
{
    HeapNode **data; // Actual MinHeap.
    int *pos;        // Stores the position of the vertex in data.
    int size;
} MinHeap;

// Creates and initializes a node of the heap.
HeapNode *createHeapNode(int vertex, int weight, int state)
{
```

```
    HeapNode *newNode = malloc(sizeof(HeapNode));
    newNode->vertex = vertex;
    newNode->distance = weight;
    newNode->state = state;
    return newNode;
}

// Allocates memory for a Minheap and returns it.
MinHeap *createMinHeap(int nodes)
{
    MinHeap *pq = (MinHeap *)malloc(sizeof(MinHeap));
    pq->data = malloc(sizeof(HeapNode *) * nodes);
    pq->pos = calloc(nodes, sizeof(int));

    pq->size = 0;
    return pq;
}

// Swap 2 integers.
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Moves an element at index 'idx' up in the heap.
void heapifyUp(MinHeap *pq, int idx)
{
    while (idx > 0)
    {
        int parent = (idx - 1) / 2;
        // If current element is smaller than its parent, swap the
           nodes in the heap array.
        if (pq->data[idx]->distance < pq->data[parent]->distance)
        {
            HeapNode *temp = pq->data[idx];
            pq->data[idx] = pq->data[parent];
            pq->data[parent] = temp;

            // Also swap the elements in the pos array.
            swap(&pq->pos[2 * pq->data[idx]->vertex + pq->data[idx
               ]->state],
                 &pq->pos[2 * pq->data[parent]->vertex + pq->data[
                    parent]->state]);

            idx = parent;
        }
        else
        {
```

```c
                break;
            }
        }
}

// Moves an element at index 'idx' down to restore the MinHeap.
void heapifyDown(MinHeap *pq, int idx)
{
    while (1)
    {
        int left = 2 * idx + 1;
        int right = 2 * idx + 2;
        int smallest = idx;

        if (left < pq->size && pq->data[left]->distance < pq->data[smallest]->distance)
        {
            smallest = left;
        }
        if (right < pq->size && pq->data[right]->distance < pq->data[smallest]->distance)
        {
            smallest = right;
        }

        if (smallest != idx)
        {
            // Swap the nodes in the heap array.
            HeapNode *temp = pq->data[idx];
            pq->data[idx] = pq->data[smallest];
            pq->data[smallest] = temp;

            // Update positions: the vertices have swapped indices.
            swap(&pq->pos[2 * pq->data[idx]->vertex + pq->data[idx]->state],
                 &pq->pos[2 * pq->data[smallest]->vertex + pq->data[smallest]->state]);

            // Continue heapifying down from the new index.
            idx = smallest;
        }
        else
        {
            break;
        }
    }
}

// Removes the smallest (root) element from the priority queue and returns it.
```

```c
HeapNode *popMinHeap(MinHeap *pq)
{
    if (pq->size == 1)
    {
        HeapNode *node = pq->data[0];
        pq->data[0] = NULL; // Clear the pointer to avoid duplicate
            free.
        pq->size--;
        return node;
    }
    HeapNode *rootNode = pq->data[0];
    HeapNode *lastNode = pq->data[pq->size - 1];

    // Move the last element to the root and clear the old index.
    pq->data[0] = lastNode;
    pq->data[pq->size - 1] = NULL; // Clear duplicate reference.
    pq->size--;

    pq->pos[lastNode->vertex * 2 + lastNode->state] = 0;
    pq->pos[rootNode->vertex * 2 + rootNode->state] = -1;

    // Heapify downward to restore the MinHeap property.
    heapifyDown(pq, 0);
    return rootNode;
}

int isEmpty(MinHeap *pq)
{
    return pq->size == 0;
}

// Free the priority queue's variables.
void destroyMinHeap(MinHeap *pq, int nodes)
{
    if (pq)
    {
        // free the data array
        for (int i = 0; i < nodes; i++)
        {
            if (pq->pos[i] != -1)
            {
                free(pq->data[i]);
            }
        }
        free(pq->data);
        free(pq->pos);
        free(pq);
    }
}

```

```c
void dijkstra(ListPointer *originalNeighbourList, ListPointer *
    reverseNeighbourList, int nodes, int *cButtons)
{
    // Helps build the best path.
    int *previous = malloc(nodes * 2 * sizeof(int));

    // Prevents from accessing deleted memory.
    int *distances = malloc(nodes * 2 * sizeof(int));

    MinHeap *minHeap = createMinHeap(nodes * 2);
    minHeap->size = 2 * nodes;

    // Initialization step.
    int id;
    for (int v = 0; v < nodes; v++)
    {
        for (int s = 0; s < 2; s++)
        {
            // Each vertex can be reversed, s(tate) keeps this
                change.
            id = 2 * v + s;

            if (v == 0 && s == 0)
            {
                distances[id] = 0;
            }
            else
            {
                distances[id] = INT_MAX;
            }
            minHeap->data[id] = createHeapNode(v, distances[id], s)
                ;
            minHeap->pos[id] = id;

            // No chamber has yet been visited.
            previous[id] = -1;
        }
    }

    while (!isEmpty(minHeap))
    {
        HeapNode *node = popMinHeap(minHeap);
        int u = node->vertex;
        int s = node->state;
        // Unique id determined by vertex and state.
        int id = 2 * u + s;

        // Press the button if possible, relaxing distance to
            button press to 0.
        if (cButtons[u])
```

```
        {
            int idToggle = 2 * u + (1 - s);
            if (distances[id] < distances[idToggle])
            {
                distances[idToggle] = distances[id];
                previous[idToggle] = id;
                minHeap->data[minHeap->pos[idToggle]]->distance = distances[idToggle];
                // After relaxing, restore the heap.
                heapifyUp(minHeap, minHeap->pos[idToggle]);
            }
        }
        // Current neighbours depend on state of maze.
        ListPointer *activeGraph = (s == 0) ? originalNeighbourList : reverseNeighbourList;

        // Traverse through the neighbours, relaxation step.
        for (ListNode *i = activeGraph[u]; i != NULL; i = i->next)
        {
            int v = i->vertex;
            // The state remains the same on travel
            int neighbourId = 2 * v + s;
            if (distances[id] != INT_MAX && distances[id] + i->weight < distances[neighbourId])
            {
                distances[neighbourId] = distances[id] + i->weight;
                previous[neighbourId] = id;
                minHeap->data[minHeap->pos[neighbourId]]->distance = distances[neighbourId];
                // After relaxing, restore the heap.
                heapifyUp(minHeap, minHeap->pos[neighbourId]);
            }
        }
        free(node);
    }

    // Determine the best distance at the destination, whether reversed or original maze.
    int id0 = (nodes - 1) * 2 + 0, id1 = nodes * 2 - 1;
    int bestDist = distances[id0];
    int bestState = 0;
    if (distances[id1] < bestDist)
    {
        bestDist = distances[id1];
        bestState = 1;
    }
    if (bestDist == INT_MAX)
    {
        printf("IMPOSSIBLE\n");

```

```
        free(previous);
        free(distances);
        destroyMinHeap(minHeap, nodes * 2);

        return;
    }
    printf("%d\n", bestDist);

    // Reconstruct path from (0,0) to (n-1, bestState)
    int *path = malloc(nodes * 2 * sizeof(int)), pathLen = 0;
    int current = (nodes - 1) * 2 + bestState;
    // Construct the path in reverse.
    while (current != -1)
    {
        path[pathLen] = current;
        pathLen++;
        current = previous[current];
    }
    // Print the path in order (convert to 1-based indexing).
    // Mark where a button was pressed (i.e. when state changes
       between consecutive nodes).
    int lastPrinted = -1;
    for (int i = pathLen - 1; i >= 0; i--)
    {
        int vertex = path[i] / 2;
        int state = path[i] % 2;

        // Avoid printing a button press as a step in the path by
           skipping iteration.
        if (vertex == lastPrinted)
        {
            continue;
        }
        else
        {
            printf("%d", vertex + 1);
        }
        if (i > 0)
        {
            int prevState = path[i - 1] % 2;
            // If state changed, that means a button press occurred
               at this chamber.
            if (state != prevState)
            {
                printf(" R");
            }
        }
        printf("\n");
        lastPrinted = vertex;
    }
```

```c
    free(previous);
    free(distances);
    free(path);
    destroyMinHeap(minHeap, nodes * 2);
}

int main()
{
    int n, m;
    scanf("%d %d", &n, &m);

    // Arrays of lists, entry i contains all neighbors of node i.
    ListPointer *originalNeighbourList = malloc(n * sizeof(
        ListPointer));
    ListPointer *reverseNeighbourList = malloc(n * sizeof(
        ListPointer));

    // Initialize original graph and reversed graph.
    for (int i = 0; i < n; i++)
    {
        originalNeighbourList[i] = NULL;
        reverseNeighbourList[i] = NULL;
    }

    int *cButtons = calloc(n, sizeof(int));

    // Read the input.
    int temp;
    scanf("%d", &temp);
    while (temp != -1)
    {
        cButtons[temp - 1] = 1;
        scanf("%d", &temp);
    }

    int src, dst, weight;
    for (int i = 0; i < m; i++)
    {
        scanf("%d %d %d", &src, &dst, &weight);
        // Vertices are stored in base 0 in the data structures.
        addEdge(originalNeighbourList, src - 1, dst - 1, weight);
        addEdge(reverseNeighbourList, dst - 1, src - 1, weight);
    }

    dijkstra(originalNeighbourList, reverseNeighbourList, n,
        cButtons);

    // Free the dynamically allocated variables.
    free(cButtons);

```

```
    for (int i = 0; i < n; i++)
    {
        ListPointer current = originalNeighbourList[i];
        while (current)
        {
            ListPointer temp = current;
            current = current->next;
            free(temp);
        }
        current = reverseNeighbourList[i];
        while (current)
        {
            ListPointer temp = current;
            current = current->next;
            free(temp);
        }
    }

    free(originalNeighbourList);
    free(reverseNeighbourList);

    return 0;
}
```