Assignment 5:"Maze"
Programming report

s5885671 and s5962021

Algorithms and Data Structures in C (2024-2025)

Problem description

General:
The goal is to write a C program that finds the shortest path through a special type of maze called the *Inverto-Maze*. This maze consists of chambers connected by one-way tunnels. Some chambers contain a special button (we will henceforth denote the button as `REV`) that, when pressed, reverses the direction of all tunnels in the maze.

The program should compute the shortest path from the entrance (chamber 1) to the exit (chamber n), taking into account any reversals caused by pressing these buttons.

Input and output behavior:
The input begins with two integers: the number of chambers n and the number of tunnels m. This is followed by a list of chambers that have a reverse (`REV`) -button, ending with `-1`. Then come m lines, each containing three integers: the starting chamber a, the destination chamber b, and the length ℓ of the tunnel from a to b. All tunnels are initially one-way from a to b.

The output starts with the total length of the shortest path from chamber 1 to chamber n. Then, the sequence of chambers along the path is printed, one per line. If a button is pressed in a chamber, an `R` is printed after its number. The final chamber (chamber n) is never followed by an `R`. If no path exists from chamber 1 to n, the output should be `IMPOSSIBLE`.

1. Problem analysis

To solve the Inverto-Maze problem, we simulate the Dijkstra algorithm while tracking two possible states for each chamber: *normal* and *inverted*. Each state corresponds to the current tunnel direction of the maze. If a chamber has a reverse button, algorithm allows one to switch from the normal state to the inverted state (or vice versa) when passing through that chamber, without increasing the total distance of the path.

The algorithm operates over two graphs (original and inverted) and uses a min-heap as a priority queue. At each step, the current chamber and its state are used to determine the valid edges. Distances are updated only if a shorter path is found. The final shortest path is determined by comparing the distance to the exit chamber in both states.

algorithm ModifiedDijkstra(maze, invertedMaze)

input: original and reversed maze graphs
result: shortest path from chamber 1 to chamber n

```
 n ← number of chambers in maze
 dist[c][s] ← infinity for all chambers c and states s
 dist[1][0] ← 0 /* start from chamber 1 in normal state */
 heap ← an empty min-heap
insert (1, 0, 0) into heap /* (chamber, state, distance) */

while heap is not empty
    (c, s, d) ← pop from heap
    if (c, s) already visited then continue
    mark (c, s) as visited

    if s = 0 then edges ← maze.tunnels[c]
    else edges ← invertedMaze.tunnels[c]

    while edges not null
        neighbor ← edges.to
        newDist ← d + edges.weight
        if newDist < dist[neighbor][s] then
            dist[neighbor][s] ← newDist
            prev[neighbor][s] ← (c, s)
            insert (neighbor, s, newDist) into heap
    edges ← edges.next

    if chamber c has a reverse button then
        switchedTo ← 1 - s /* the opposite state of the current chamber */
        if d < dist[c][flipped] then
            dist[c][flipped] ← d
            prev[c][flipped] ← (c, s)
            insert (c, flipped, d) into heap

 exitDist ← minimum of dist[n][0] and dist[n][1]
if exitDist = infinity then print IMPOSSIBLE
else print exitDist and backtrack path from corresponding state
```

2. Program design

We define helper functions in multiple files to handle input reading, heap operations, and graph representation. The program uses a custom `Heap` structure to act as a priority queue for Dijkstra's algorithm, and a `Maze` structure to represent both the original and reversed tunnel graphs. Each chamber is represented by a `Chamber` struct, and the algorithm tracks progress using `NodeState` structs, which store the current state (normal or inverted), distance, and backtracking information.

The core logic is implemented in the function `findShortestPath`. It starts by initializing all node states (for both the normal and inverted versions of the maze), as well as a `visited` table and a heap. The algorithm runs a modified version of Dijkstra's algorithm, in which each chamber is considered in both of its states. At each step, the algorithm pops the node with the smallest known distance from the heap and explores its outgoing tunnels from either the original or inverted graph, depending on the current state. If a chamber has a reverse button, the state can be flipped without increasing the total path length.

The main program (`main.c`) reads the input maze using `readInput`, constructs its inverted version using `invertMaze`, and then calls `findShortestPath` to compute the shortest route to exit. After that, both maze representations are de-allocated.

Design choice. Instead of repeatedly flipping the maze each time the reverse button is considered, we chose to construct the reverse maze once in advance. This avoids the need to perform costly graph operations during path-finding and simplifies the state-flipping logic during traversal. The state of each chamber (normal or inverted) is treated as a separate node in the algorithm, allowing clean separation of logic and easier heap management. This design also aligns well with the structure of Dijkstra's algorithm, which can be extended naturally to multi-state systems.

Time complexity. Let n be the number of chambers and m the number of tunnels. Since each chamber has two states, the graph effectively has $2n$ nodes. For each connection in the maze, the algorithm checks (at most once per state) whether it leads to a shorter path to the next chamber. This results in at most `O(m)` such checks across the entire maze. Insertions and removals from the heap (which contains up to $2n$ nodes) take $\mathcal{O}(\log n)$ time. Therefore, the total time complexity of the algorithm is $\mathcal{O}(m \log n)$.

3. Evaluation of the program

We tested the program using the input file `3.in`, which contains the following input:

```
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
```

This input describes a maze with 17 chambers and 20 tunnels. Chambers 1, 7, 14, and 10 are equipped with reverse buttons. The maze includes multiple paths, cycles, and sections where reversal of tunnel directions is essential to progress.

The output produced by the program for this input was:

```
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
```

This output indicates that the shortest path from chamber 1 to chamber 17 has a total length of 133.

This result is correct. Pressing the reverse buttons is needed to go back through earlier chambers or reach places that can't be accessed otherwise. The order of chambers and the total distance match what we would expect based on how the maze is built.

We apply valgrind to check for any memory leaks with the same input. Valrind reports the following.

```
==41654== HEAP SUMMARY:
==41654==     in use at exit: 0 bytes in 0 blocks
==41654==   total heap usage: 162 allocs, 162 frees, 11,136 bytes allocated
==41654==
==41654== All heap blocks were freed -- no leaks are possible
```

```
==41654==
==41654== For lists of detected and suppressed errors, rerun with: -s
==41654== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No memory leaks are found.

Therefore, we conclude that the program correctly finds the shortest path and handles tunnel direction reversals as intended.

4. Process description

We based our solution on a modified version of Dijkstra's algorithm, adapted to handle the Inverto-Maze's unique mechanic of tunnel direction reversals. It worked well to model each chamber in two separate states—*normal* and *inverted*—and treat these as distinct nodes when computing the shortest path. This dual-state representation formed the core structure of the `findShortestPath` function.

Handling transitions between states, we constructed an inverted version of the maze at the beginning, instead of reversing the tunnel directions dynamically during the search. This allowed us to focus more on the logic inside the main loop to determine which version of the maze to traverse based on the current state. When a reverse button was encountered, we conditionally pushed the alternate state to the heap without increasing the path length.

During development, we faced several technical challenges. One of them was refactoring the code to keep responsibilities separated across multiple files, such as keeping heap operations in `Heap.c` and maze-related logic in `Maze.c`. Managing multiple custom structs at once—like `Chamber`, `NodeState`, `Tunnel`, and `Maze`—was sometimes complex, especially when passing pointers between functions. Debugging memory issues and using tools like `valgrind` to detect and fix leaks was also a significant part of our workflow.

By working on this project, we learned how to extend classical algorithms like Dijkstra's to more complex, state-based problems. We also became more confident in managing memory and struct-based data in C, and gained practical experience with modular design and debugging across multiple source files.

5. Conclusions

The program solves the problem of finding the shortest path through the Inverto-Maze by extending Dijkstra's algorithm to handle dynamic tunnel reversals. It does so in a clean and effective way by treating each chamber-state combination as a distinct node. The program is efficient; its time complexity is bounded by $\mathcal{O}(m \log n)$, which is suitable given the need to explore multiple states per chamber.

6. Appendix: program text

We only include the `helpers.c` file in the report, as it contains the main logic of the program and implements the modified Dijkstra's algorithm used to solve the problem.

`helpers.c`

```c
#include "helpers.h"

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

#include "Heap.h"

#define INF (1 << 30)

Maze *readInput() {
  // Initialize variables and get the values of n, m.
  int numberOfChambers = 0, numberOfTunnels = 0,
      chamberWithInvertButton = 0, weight = 0, from = 0, toIndex = 0;
  scanf("%d %d", &numberOfChambers, &numberOfTunnels);

  // Create the Maze.
  Maze *maze = createMaze(numberOfChambers, numberOfTunnels);

  // Add the Chambers to the Maze.
  for (int chamberIndex = 1; chamberIndex <= numberOfChambers;
      chamberIndex++) {
    Chamber *newChamber = createChamber(chamberIndex, 0);
    maze->chambers[chamberIndex - 1] = newChamber;
  }

  // Read which Chambers that have an invert button.
  scanf("%d", &chamberWithInvertButton);
  while (chamberWithInvertButton != -1) {
    maze->chambers[chamberWithInvertButton - 1]->hasInvertButton =
        1;
    scanf("%d", &chamberWithInvertButton);
  }

  // Read the connections between the Chambers and the Tunnels.
  for (int tunnel = 0; tunnel < numberOfTunnels; tunnel++) {
    scanf("%d %d %d", &from, &toIndex, &weight);
    Chamber *toChamber = maze->chambers[toIndex - 1];
    addTunnel(maze, weight, from - 1, toChamber);
  }

  return maze;
}

// Function to free reserved memory for most of the created
   instances used for Dijkstra's Algorithm.
static void freeMemory(int **visited, int visitedLength, Heap *heap
    , NodeState **nodes, int nodesLength) {
  for (int i = 0; i < visitedLength; i++) {
    free(visited[i]);
  }
```

```
46    free(visited);
47    freeHeap(heap);
48    for (int node = 0; node < nodesLength; node++) {
49      free(nodes[node]);
50    }
51    free(nodes);
52  }
53
54  void findShortestPath(Maze *maze, Maze *invertedMaze) {
55    int n = maze->numberOfChambers;
56    int totalNodes = n * 2;
57    int **visited = calloc(n, sizeof(int *));
58
59    for (int i = 0; i < n; i++) {
60      visited[i] = calloc(2, sizeof(int));
61    }
62
63    NodeState **nodes = malloc(totalNodes * sizeof(NodeState *));
64    for (int i = 0; i < n; i++) {
65      // Normal Maze NodeState instances.
66      nodes[i * 2] = malloc(sizeof(NodeState));
67      nodes[i * 2]->chamber = maze->chambers[i];
68      nodes[i * 2]->state = 0;
69      nodes[i * 2]->distance = (i == 0) ? 0 : INF;
70      nodes[i * 2]->prev = NULL;
71
72      // Inverted Maze NodeState instances.
73      nodes[i * 2 + 1] = malloc(sizeof(NodeState));
74      nodes[i * 2 + 1]->chamber = maze->chambers[i];
75      nodes[i * 2 + 1]->state = 1;
76      nodes[i * 2 + 1]->distance = INF;
77      nodes[i * 2 + 1]->prev = NULL;
78    }
79
80    // Create a Heap instance to represent the To-Do List for
         Dijkstra's Algorithm and initialize it.
81    Heap *heap = createHeap();
82    insertNodeState(heap, nodes[0]);
83
84    // Perform Dijkstra's Algorithm.
85    while (!isHeapEmpty(heap)) {
86      NodeState *current = popMinNodeState(heap);
87      int curIndex = current->chamber->index - 1;
88      if (visited[curIndex][current->state]) {
89        continue;
90      }
91      visited[curIndex][current->state] = 1;
92
93      Tunnel *tunnel = (current->state == 0) ? maze->tunnels[curIndex
         ] : invertedMaze->tunnels[curIndex];
```

```
94      while (tunnel) {
95        int neighborIndex = tunnel->to->index - 1;
96        NodeState *neighbor = nodes[neighborIndex * 2 + current->
              state];
97        int newDistance = current->distance + tunnel->weight;
98        if (newDistance < neighbor->distance) {
99          neighbor->distance = newDistance;
100         neighbor->prev = current;
101         insertNodeState(heap, neighbor);
102       }
103       tunnel = tunnel->next;
104     }
105
106     if (current->chamber->hasInvertButton) {
107       NodeState *switched = nodes[curIndex * 2 + (1 - current->
              state)];
108       if (current->distance < switched->distance) {
109         switched->distance = current->distance;
110         switched->prev = current;
111         insertNodeState(heap, switched);
112       }
113     }
114   }
115
116   // Get the NodeState representation of the exit Chamber.
117   int exitIndex = n - 1;
118   NodeState *exitNormal = nodes[exitIndex * 2];
119   NodeState *exitInverted = nodes[exitIndex * 2 + 1];
120   NodeState *bestExit = (exitNormal->distance <= exitInverted->
          distance) ? exitNormal : exitInverted;
121
122   // If the distance from the shortest path (bestExit->distance) is
          still INF then print IMPOSSIBLE.
123   if (bestExit->distance == INF) {
124     printf("IMPOSSIBLE\n");
125
126     // Free all reserved memory and return.
127     freeMemory(visited, n, heap, nodes, totalNodes);
128     return;
129   }
130
131   // We found the shortest path so now we have to print its length
          and the actual path.
132   printf("%d\n", bestExit->distance);
133   int count = 0;
134   NodeState **path = NULL;
135   NodeState *p = bestExit;
136
137   // Back-track the Chambers we went through and store them to the
          path list.
```

```
138   while (p) {
139     path = realloc(path, (count + 1) * sizeof(NodeState *));
140     path[count++] = p;
141     p = p->prev;
142   }
143
144   // Print the path list.
145   for (int i = count - 1; i >= 0; i--) {
146     // Check if we pressed the invert button at the current
          NodeState Chamber.
147     if (i > 0 && path[i]->chamber->hasInvertButton && path[i]->
          state != path[i - 1]->state) {
148       printf("%d R\n", path[i]->chamber->index);
149     } else if (i == count - 1 || !(path[i]->chamber->
          hasInvertButton && path[i]->state != path[i + 1]->state)) {
150       printf("%d\n", path[i]->chamber->index);
151     }
152   }
153
154   // Free all reserved memory.
155   free(path);
156   freeMemory(visited, n, heap, nodes, totalNodes);
157 }
```