Assignment 2

COP290 Simulation Report

Sayam Sethi 2019CS10399
Mallika Prabhakar 2019CS50440

supervised by
Prof. Rijurekha Sen

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

June 2021

Contents

Chapter 1

Situation

1.1 Story

Continuing the story from Assignment 2 Task 1, the monopolistic ruling body realised that 1v1 matches are slow and the rate of passing is still high (50%). The material for vaccine generation was depleting rapidly and they were criticised for killing methods hence they decided to make a different test for analysing worth without involving deaths. They needed lesser percentage of people pass and find the physically and mentally strong people to give vaccines to.

They finally decided on a normal maze game but with some twists. The maze was constructed on a rocky terrain with different sections of maze having different difficulty of traversal. Entire maze will be filled with staining gas (Non toxic gas which gets accumulated in lungs). The participants were given a certain concentration of the staining-gas before the game started. There will be one time accessible anti-gas chambers which pump out certain quantity of staining gas from their lungs. The person who has the least content of staining gas in lungs and comes out the fastest is eligible for vaccine and others are barred from getting the same. In case of a tie regarding quantity of staining gas, the one who came out earlier will be eligible to obtain the vaccine.

To check mental strength, They give the participants a map marked with terrain difficulty and anti-gas chamber functioning stats along with start and end gate locations a few minutes before the game starts.

1.2 Problem statement

Given a map of the maze denoted with terrain difficulty, anti-gas chamber locations and values and starting and ending points, find and traverse the most efficient route fastest i.e. route which results in least staining gas concentration in lungs. Since each participant is given the map of the maze only minutes before they have to enter, being fast is crucial.

This results in us creating two different algorithms, one with large complexity but the best answer and a faster heuristic approach.

1.3 Assumptions

Following are the general assumptions taken for the development of algorithms.

- Players can't jump over walls or take illegal routes
- Opponents act fairly and do not affect the game play of others hence are not included in the simulation
- The movement speed is considered to be same for all
- Each anti gas chamber can be accessed once

1.4 Mathematical Interpretation of the Problem Statement

An $n \times n$ sized maze is given with a total of k *special nodes*. Each node i has a weight w_i, with the *special nodes* having negative weights and every other node having positive weights. Each Player starts at the entry gate (S). The motive is to find the optimal path to the exit gate (E) minimising the value

$$\sum_{i \in P} c_i \tag{1.1}$$

where P is the multi-set of nodes in the path taken from S to E and $c_i = 0$ for all occurrences of $i \in K$ (where K is the set of *special nodes*) which is not the first one.

Chapter 2

Brute Force Approach

2.1 Approach discussion

The simplest brute force approach is to check all possible situations, i.e., checking over the number of special vertices visited and the order of visiting. Formally stating, the algorithm finds the best path by iterating over all permutations for each element of the power set of the k vertices.

The pre-computation would require computing the shortest path for all *special* pairs. This can be easily done using the Floyd-Warshall Algorithm. However, there is a possibility of having cycles with negative path lengths. This can be resolved by first computing the path lengths assuming that the special vertices all have 0 weights and then adding (the negative) weights of the special vertices j for all ordered $distance(i, j)$ pairs. The correctness has been discussed below.

The power set can be easily mapped using a bitmask. However, when ordering is introduced, we will need to check all permutations of each mask. This can be efficiently done using a recursive strategy (discussed in the pseudo-code section).

Once we are done with the pre-computation and have decided on a systematic iteration over all possibilities, all that remains is computing the cost of the path for each ordering of each mask which can be easily done using the distance matrix computed earlier.

2.2 Pseudo-code

2.2.1 buildGraph

```
// process the graph from the maze
// initialise the distances (stored as umap<int, umap<int, int>>)
buildGraph(maze, distances):
  // iterate over all nodes in the maze
  for (node: maze):
    // add an edge for all neighbours to the current node
    for (neighbour: neighbours(node)):
      // if it is a special node, add an edge of weight zero
      // (since negative cycles can exist)
      if (isSpecial(neighbour)):
        addDirectedEdge(node, neighbour, 0)
        distances[node][neighbour] = 0
      else:
        addDirectedEdge(node, neighbour, weight[neighbour])
        distances[node][neighbour] = weight[neighbour]
```

2.2.2 modifiedFloydWarshall

```
// perform regular Floyd-Warshall; then update relevant distances
modifiedFloydWarshall(distances, specialNodes):
  regularFloydWarshall(distances)
  for (node1: specialNodes):
    distances[S][node1] += weight[node1]
    for (node2: specialNodes):
      if (node1 != node2):
        distances[node1][node2] += weight[node2]
```

2.2.3 enumerateAllPermutations

```
// enumerates all possibities and computes the path length
enumerateAllPermutations(distances, specialNodes):
  bestDist = INF, ans = []
  for (mask: [0, 1 << k)):
    chosenSpecialNodes = []
    for (i: [0, k)):
      if mask & (1 << i):
        chosenSpecialNodes.add(specialNodes[i])
    sort(chosenSpecialNodes)
    for (permutation: chosenSpecialNodes):
      dist = 0, prevNode = startNode
      for (node: permutation):
        dist += distances[prevNode][node]
        prevNode = node
      dist += distances[prevNode][exitNode]
      if (dist < bestDist):
        bestDist = dist
        ans = permutation
  return (bestDist, ans)
```

2.3 Proof of Correctness

Claim 2.3.1 (property of pairwise distances)**.** *The **modified Floyd-Warshall** algorithm computes the correct distances for all pairs of nodes in* $\{startNode, endNode\} \cup K$ *(under the assumption that the path is a **straight path**).*

(**Straight path**: A straight path is a path from a node in the set $\{startNode, endNode\} \cup K$ to another node in the same set which has contribution from every other *special node* equal to 0 when computing the cost of the path)

Proof. We know that the weights of the *special nodes* is negative only when we first encounter it.
Since we only consider *special paths*, the contribution of every other *special node* to the path length is 0, which is what we use during the *Floyd-Warshall algorithm*. Additionally, the destination *special node* always contributes to the path length (exactly once) and thus, the shortest path length can be computed as:

$$\begin{aligned} &min(distance(u, neighbour)) + weight(specialNode) \\ &= FloydWarshallDistance(u, specialNode) + weight(specialNode) \end{aligned} \tag{2.1}$$

Hence, the correctness of the *modified Floyd-Warshall* algorithm is proved. □

Claim 2.3.2 (correctness of enumerateAllPermutations)**.** *The straight paths computed as defined in **claim 2.3.1** can be used to compute the best path in the function **enumerateAllPermutations**.*

Proof. The function *enumerateAllPermutations* considers all permutations of every subset of the powerset of the special nodes.
Consider the ordered set B, which is the set of the the special nodes that are visited in the path with the least cost.
It can be shown that the shortest path from S to E which traverses through the nodes in B can be constructed using *straight path* between S and the first element of B, *straight paths* between every consecutive element in B and the *straight path* between the last element of B and E.
Conversely, assume that none of the path between any two consecutive points is not the straight path. Using *claim 2.3.1*, the shortest path between the given two nodes is equal to the path length computed by the modified algo since no other special node is visited between the two (otherwise the two nodes would not have been consecutive in the order of visiting). Therefore, the length of the path between these two nodes is larger than the shortest possible path and hence the total path length from S to E would be larger. This is a contradiction to the fact that none of the path is a *straight path*. Hence, the most optimal path is computed using *enumerateAllPermutations*. □

Chapter 3

Faster Solution using Heuristics

3.1 Observations

The optimal solution has a pre-computation which is very slow and computes a lot of redundant distance values which are never accessed. Only distances from *start node* and *special nodes* are required. Additionally, a lot of the permutations which are processed are clearly suboptimal and their computation should be ignored. For example, going to the farthest *special node* and then coming back to the nearest *special node*, wrt the *start node*.

The heuristic that will be discussed aims at optimising primarily on these two aspects. Instead of Floyd-Warshall algorithm, multiple runs of Dijkstra algorithm are done. However, since Dijkstra only works on graphs with non-negative weights, the weights of the special vertices are assumed to be 0 when performing the algorithm.

3.2 Heuristic Algorithm

The optimised algorithm uses a *greedy* approach by choosing which *special node* to move to next, or directly move to the *exit node*, using the shortest path. The algorithm only considers those *special nodes* which can reduce the path length on visiting, i.e., it considers nodes only if $distance(currentNode, specialNode) + distance(specialNode, exitNode) < distance(currentNode, exitNode)$. Out of all these nodes, it chooses the node which is the closest and if there are multiple closest nodes, it chooses the one which offers the least reduction in the cost, if no other vertex was visited. Effectively, the algorithm looks ahead by a single step.

3.3 Pseudo-code

3.3.1 buildGraph

```
// process the graph from the maze
buildGraph(maze):
  // iterate over all nodes in the maze
  for (node: maze):
    // add an edge for all neighbours to the current node
    for (neighbour: neighbours(node)):
      // if it is a special node, add an edge of weight zero
      // (since Dijkstra doesn't work with negative edges)
      if (isSpecial(neighbour)):
        addDirectedEdge(node, neighbour, 0)
      else:
        addDirectedEdge(node, neighbour, weight[neighbour])
```

3.3.2 computeDistances

```
// compute distances from startNode and all specialNodes
// to all other specialNodes and exitNode
computeDistances(graph, startNode, specialNodes):
  // using a hash map to store distances from each source
  // Dijkstra returns the distance vector
  distances[startNode] = Dijkstra(startNode, graph)
  for (node: specialNodes):
    distances[node] = Dijkstra(node, graph)
```

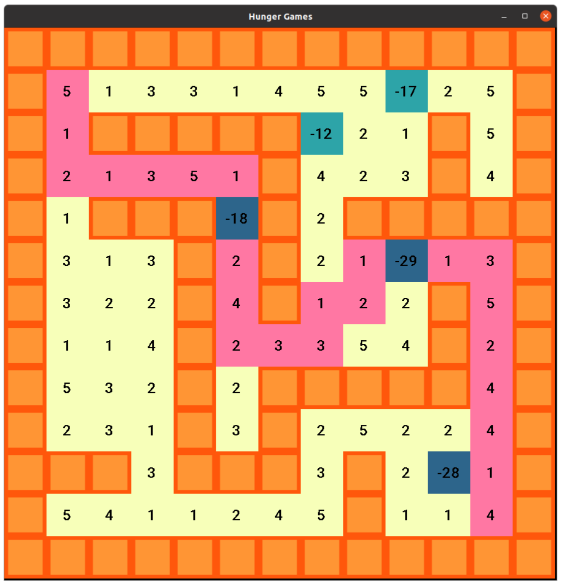
3.3.3 findBestPath

```
findBestPath(graph, distances, specialNodes):
  currentNode = startNode
  pendingSpecialNodes = specialNodes
  while (currentNode != exitNode):
    D = distances[currentNode][exitNode]
    closest = D
    bestCost = D
    bestNode = exitNode
    for (node: pendingSpecialNodes):
      d = distances[currNode][node]
      tot = d + distances[node][exitNode] + weights[node]
      if (tot < D):
        if (d < closest || (d == closest && tot < bestCost)):
          closest = d
          tot = bestCost
    pendingSpecial.remove(bestNode)
    currNode = bestNode
```

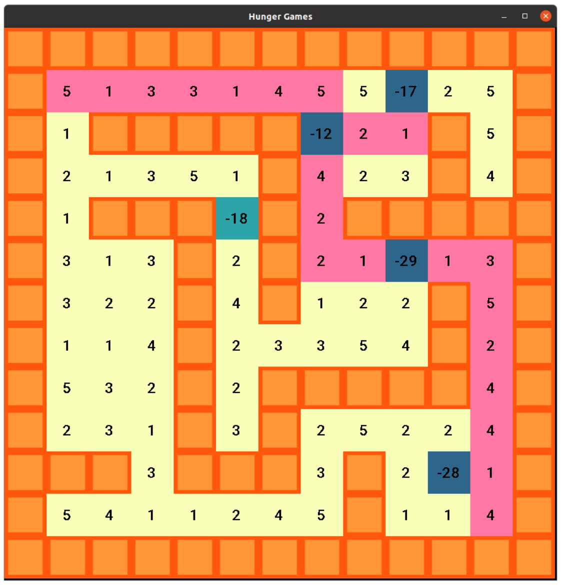
Chapter 4

Simulation Display

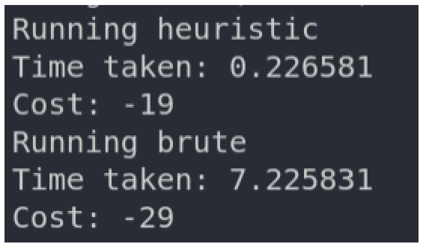
4.1 Heuristic Slightly sub-Optimal



(a) Path chosen by heuristic algorithm (darker vertices are visited); runtime = 0.22s; cost = -19



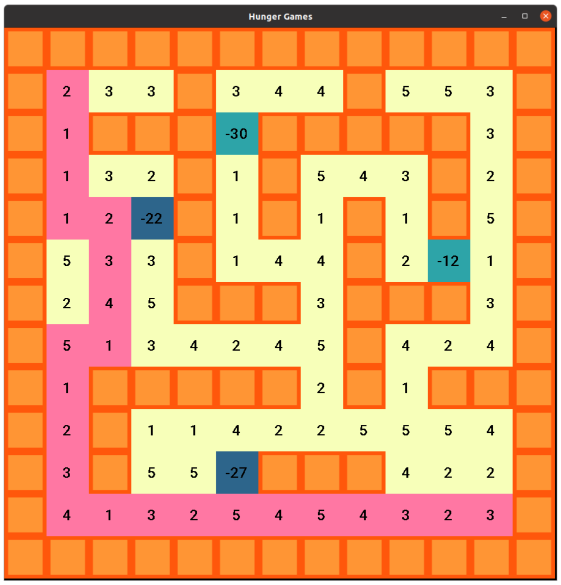
(b) Path chosen by brute force algorithm (darker vertices are visited); runtime = 7.22s; cost = -29



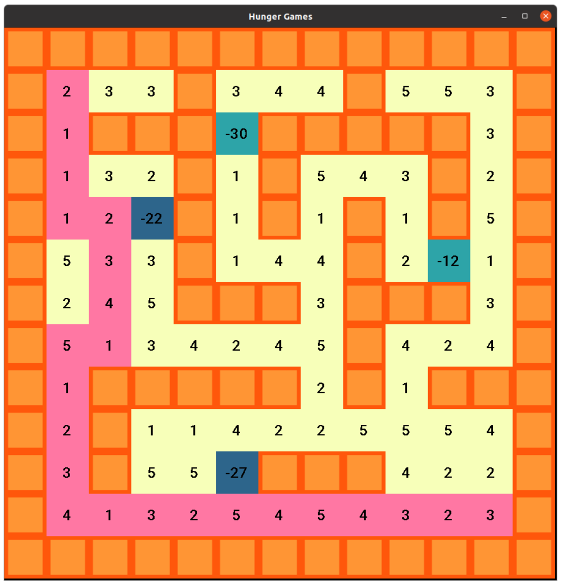
```
Running heuristic
Time taken: 0.226581
Cost: -19
Running brute
Time taken: 7.225831
Cost: -29
```

Figure 4.2: Runtime and cost

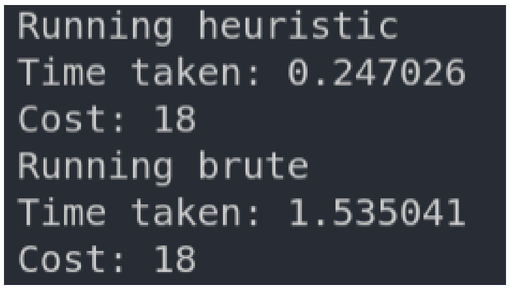
4.2 Heuristic Optimal



(a) Path chosen by heuristic algorithm (darker vertices are visited); runtime = 0.24s; cost = 18



(b) Path chosen by brute force algorithm (darker vertices are visited); runtime = 1.53s; cost = 18



```
Running heuristic
Time taken: 0.247026
Cost: 18
Running brute
Time taken: 1.535041
Cost: 18
```

Figure 4.4: Runtime and cost

Chapter 5

Analysis

We are considering the following runtimes by not adding the time taken to generate a maze and set special vertices along with assigning weight to individual vertex.

For completeness, following are the time and space complexities of the functions called before calling the brute force and heuristic approaches:

Function name	time complexity	space complexity
generateSpecialVertices	$O(k)$	$O(1)$
assignWeights	$O(n^2)$	$O(1)$
backtrack	$O(n^2)$	$O(1)$
generateRandomMaze	$O(n^2)$	$O(n^2)$
removeWalls	$O(n^2)$	$O(1)$
generateRandomWalls	$O(n^2)$	$O(n^2)$

5.1 Runtime analysis for brute force approach

There are three functions in the brute force approach which are called in succession. Thus the total time complexity is sum of the individual time complexities.

$$\begin{aligned}
&T(buildGraph) + T(modifiedFloydWarshall) + T(enumerateAllPermutations) \\
&= O(n^4) + O(n^6 + k^2) + O(\sum_{i=0}^{k} \binom{k}{i} \times (i \times log(i) + i! \times i)) \\
&= O(n^4) + O(n^6 + k^2) + O(\sum_{i=0}^{k} \binom{k}{i} \times (i! \times i)) \\
&= O(n^6 + k! \times k)
\end{aligned} \tag{5.1}$$

5.2 Runtime analysis for Heuristic algorithm

There are three functions in this approach as well which are called in succession. Thus the total time complexity is sum of the individual time complexities.

$$\begin{aligned} & T(buildGraph) + T(computeDistances) + T(findBestPath) \\ & = O(n^4) + O(k \times T(Dijkstra)) + O(k^2) \\ & = O(n^4) + O(k \times (n^2 + n^4 \times log(n^2))) + O(k^2) \\ & = O(k \times n^4 \times log(n) + k^2) \end{aligned} \tag{5.2}$$

5.3 Comparison of the Two Solutions

Insert trade-off analysis between error and runtime

n	k	Brute Force			Heuristic Algorithm		
		time taken	cost	chosen cells	time taken	cost	chosen cells
17	8	4.187510	−4	5	0.009284	−4	5
17	8	5.511454	−44	7	0.010593	−26	4
15	3	1.364807	33	2	0.003830	33	2
15	3	1.443526	49	1	0.004981	52	3
11	7	0.300426	−22	6	0.004489	−22	6
11	7	0.319058	−53	6	0.004724	−45	4
15	4	1.389357	35	2	0.005482	35	2
15	4	1.474679	−2	3	0.004966	−2	3
9	6	0.072333	−63	6	0.002307	−44	4
17	2	3.298533	86	0	0.003558	86	0

5.4 Observations

Following observations are made with respect to both the algorithms:

- Time take to find the path using brute force algorithm is very large compared to the heuristic approach
- Brute force always provides the path of least cost but at a heavy price of time
- Heuristic approach does not consider the fact that the path might contains another special vertex hence it provides sub optimal solutions at times
- Time complexity of Brute force algorithm is: $O(n^6 + k! \times k)$
- Time complexity of Heuristic algorithm is: $O(k \times n^4 \times log(n) + k^2)$
- For a large n:k ratio, brute force and heuristic algorithm give similar paths as solution with heuristic algorithm being way faster