Maze Simulation

Nishant Kumar, 2019CS50586
Mihir Meena, 2019CS10370

July 2021

1 Introduction

We have a friend named Jack doing a part-time job at the busiest pizza house in the city. He has to deliver pizzas to the customers and report back to the pizza house. He wants to do these deliveries in the shortest possible time and get back to his home to complete his college assignments. But he is new in the city and worried about his study. So, we decided to help him by making a simulation of a path that takes close to minimum time to complete his job. He can also see his current location on the simulation.

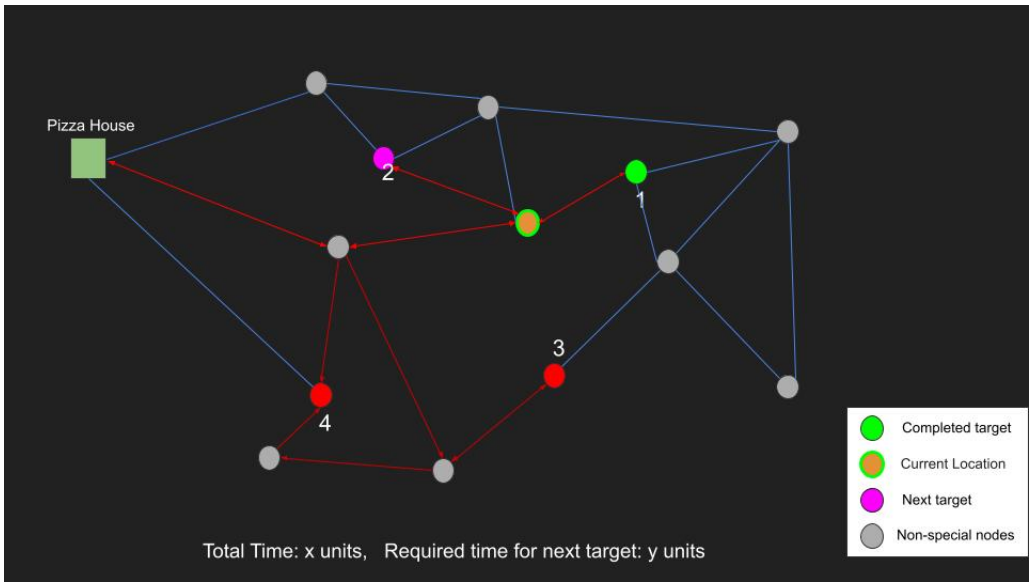
2 Map the city

The city is like a maze. So, we are dividing the city into small grids. Each grid will represent the vertices of a graph, G, and each grid is connected to its adjacent four grids. The weight of edges of the graph between the adjacent grid will represent the time taken to move from one grid to another. If there is no path between the adjoining grid, then the edge's weight will be ∞. So, all the edges having non-negative weight represents a path in maze. Each delivery location will be treated as a special vertex. Our task is start from pizza house(S) and find the shortest possible walk that visits each special vertices and then returns to pizza house(S). We represent our graph G as an **adjacency list**.

3 Simulation

We will simulate the map of the city along with the delivery points and the current location. In the simulation, the pizza house and all the delivery location will be visible. The route we get from our algorithm which he has to be traversed in order to complete his task as soon as possible. A timer will be displayed which shows the expected time to reach next destination point from his current location and a timer which says the total time required to complete the task. All the completed target will be shown in different colours. Next target will be visible in different colour. Simulation will also show the current location

of the user as shown in the figure below.



4 Algorithm and data structure

We have a graph, G(V, E) where V represents set of all the vertices of the graph and E represents set of all the edges of the graph. Let $V_R \subset V$ be the set of all the special vertices to be visited. This problem can be map as traditional Steiner tree problem. We can use greedy approach of steiner TSP to get find the walk close to optimal solution. In greedy approach, we start with initial node $v_1 \in V_R$ (pizza house) to start the walk. Let N be the **unordered_set** of visited nodes and P be the walk stored in a **linked list**. We add v_i to N and P and follow the approach in the pseudo code. In pseudo cose, we set the current node to our start node v_1. The while loop is performed until all the special nodes have been visited. At each iteration, the next node to be visited is set to closest among all yet unvisited special nodes. The closest node and its shortest path P' is calculated by Dijkstra's algorithm. The set N, path P and the current node is updated futher in the pseudo code. After completing the loop, we have calculated the shortest path from current to initial node i and updated the path P. The required path is returned at the end of the code.

Start with initial node $v_1 \in V_R$
$N \leftarrow \{v_1\}$
$P \leftarrow \{v_1\}$
current node $\leftarrow v_1$
while $N \neq V_R$ do
next node $\leftarrow$ closest node to current node in $V_R \setminus N$ (can be found using Dijkstra's Algorithm)
$P' \leftarrow$ shortest path form current node to next node
$P \leftarrow P \oplus P'$
end while;
$P' \leftarrow$ shortest path form current node to initial node v_i
$P \leftarrow P \oplus P'$
return P

In this way, we can have a walk $W = (\ v_1, v_2, ..., v_k\ )$ which is pair wise optimal. So, the problem is to determine the order in which special nodes should be visited to find the optimal walk. But there are $|V_R|!$ combinations possible. So, we will just find take walk, W_m = min_time_taken(W_1, W_2,.....,W_r), where W_i is a walk from above approach whose initial node is v_i. W_m will be close to the optimal solution and computation is also feasible.

4.1 Dijkstra's Algorithm for the shortest path

In Dijkstra's, we will use **priority queue** which stores all the vertices and priority. Each vertex in the priority queue will have its priority and pointer of its parent.

- Initialize our priority queue, source vertex will have priority 0 and its parent NULL. All the other vertices will have priority of infinity and parent NULL.
- While all the special vertex are not processed and dequeued from the priority queue and marked as visited, the vertex with the lowest priority will get processed.
- Dijkstra's explores all the adjacent vertices and edges that connect the dequeued vertex with it's adjacent vertices. We skip the adjacent vertices which is already visited. We will compare the priority of the adjacent vertex with sum of edge weight and the priority of the current vertex. If priority of a adjacent vertex is more then we will update the priority to sum of edge weight and the priority of the current vertex and parent to current vertex of the adjacent vertex.
- In this way, priority of each special vertex will denote the minimum time taken to travel from source vertex to the special vertex. We can select a vertex $v \in V_R \setminus N$ which takes minimum time. We will create a linked list which stores the walk from source to the special vertex which takes minimum time and return this linked list.

We will implement priority queue with a **Fibonacci heap** instead of min heap to improve time complexity. Fibonacci heap takes O(1) time for both insertion and decreasing priority operation and min-heap tales O($logn$).

**All the data structures used are in bold.*

5 Complexity and Analysis

5.1 Time Complexity and Runtime Analysis

We know delete operation and decrease priority operation in Fibonacci heap takes O($logn$) and O(1) time respectively.
In Dijkstra's algorithm, we have to dequeue each vertex from the priority queue which takes total O($|V|$log $|V|$) time and we have decrease priority 2E times (sum of outdegree of all the vertices) which takes O(E) time. So, time complexity of Dijkstra's algorithm is O($|V|log|V|$ + E). Our greedy algorithm for steiner TSP repeating Dijkstra's $|V_R|$ times. Hence, time complexity of greedy approach will be O($|V_R|$($|V|$log $|V|$ + E)). If we select minimum time taken from r walks discussed above then it will increase our time complexity to O(($|V_R|$)^2($|V|log|V|$ + E)) but our solution will be more close to optimal solution.
If we store the solution of each Dijkstra's we have done then we can decrease time complexity of the algorithm to some extent but that will require more space.

5.2 Hardness analysis of Problem

We have implemented a greedy algorithm which gives approximate solution to Steiner TSP. Here we will prove Steiner tree problem in graphs is **NP-complete**.

- First, we show that Steiner Tree Problem, π is in NP.
- Chose a known NP-complete problem π'.
- Construct a transformation f from π' to π.
- Prove that f is a polynomial transformation.

(a) Steiner Tree is in NP
Let an undirected graph G = (V, E). A subset $V_R \subset V$, called terminal nodes. A number k $\in \mathbb{N}$.
P: There is a subtree of G that includes all the vertices of V_R and that contains at most k edges.
Assume <G, V_R, k> $\in$ ST ans P(<G, V_R, k>) is true. In this case, we have a hypothetic solution T $\subset$ G, we can check in polynomial time that:

- T is really a tree.

- The tree T touches all the terminals specified by the set V_R.
- The number of edges used by the tree is no more than k.

(b) Known NP-complete problem
Exact cover by 3-sets problem is a well known and mentioned among the basic NP-complete problems. Let a finite set X with $|X| = 3q$. A collection C of 3-element subsets of X, C = $\{C_1, ..., C_n\}$, $C_i \subset$ X, $|C| = 3$. We know that C contains an exact cover for X, that is,a subcollection C' $\subset$ C such that every element of X occurs in exactly one member of C'.

(c) We can have a transformation f from π' to π which can be done in polynomial time as proved in paper **Steiner Tree NP-completeness Proof**

6 References

[1] A GRASP heuristic using path-relinking and restarts for the Steiner traveling salesman problem
[2] Steiner Tree NP-completeness Proof