2662. Minimum Cost of a Path With Special Roads Medium Graph Shortest Path | Heap (Priority Queue) Array

movement cost and any cost savings that can be achieved by using the special roads.

designed to find the shortest path in a weighted graph with non-negative edge weights.

graph. The specific data structures and patterns used in this implementation are:

Leetcode Link

Problem Description You are placed on a grid at a starting position defined by coordinates [startX, startY]. Your goal is to move to a target position on

the grid, specified by [targetX, targetY]. The cost of moving from any position (x1, y1) to another (x2, y2) is equal to the sum of the absolute differences of their x-coordinates and y-coordinates, i.e., $|x^2 - x^1| + |y^2 - y^1|$. In addition to the regular cost of moving on the grid, there are special roads available. These are described in a list where each

element is another list of the form [x1_i, y1_i, x2_i, y2_i, cost_i], representing a special road that connects (x1_i, y1_i) to (x2_i, y2_i) at a given cost_i. You can use these special roads multiple times if needed. The challenge is to determine the minimum cost to move from the start position to the target position, considering the regular grid

Intuition To find the least costly path, we need an algorithm that can process the grid and special roads systematically while keeping track of

the costs accumulated along the way. The most suitable algorithm for this problem is Dijkstra's algorithm. It's a well-known algorithm

Dijkstra's algorithm fits our needs well because it uses a priority queue to explore nodes (positions) that have the smallest known

choosing the least costly option to explore next. The default movement cost across the grid can act as the edges with a regular weight, while the paths provided by the special roads can be seen as edges with potentially lower weights. The main difference here is that Dijkstra's algorithm typically operates on

cost to reach from the start. It updates the cost to reach neighboring nodes as it progresses, and it's greedy in nature, always

graphs with fixed nodes and edges, whereas in this problem, the edges are dynamic, as movement is not restricted to predefined connections between points. By using a modified version of Dijkstra's algorithm, we can push into the priority queue the cost to move from the current position to

every position that can be reached by a special road, while also considering the cost to move directly towards the target position from our current position. This approach systematically evaluates all possible moves (both regular and special), keeping track of visited positions to prevent circular routes and redundant calculations. In summary, the intuition is to use a priority queue to greedily accumulate the least cost of reaching the adjacent positions or the

target, considering the cost of normal moves and special roads, until we can determine the minimum cost required to reach the target. **Solution Approach**

current known cost from the start. In Python, this is typically implemented with a min-heap using the heapq library. · A Set for Visited Nodes: To keep track of all visited positions to prevent revisiting and recalculating costs for those positions, a set is used.

In Dijkstra's algorithm, nodes (in this case, grid positions) are visited in order based on their current lowest cost from a start position.

For each visited node, we update the costs to its adjacent nodes, which, in the context of the grid problem, are the other nodes that

A Priority Queue: This is used to maintain a set of all unvisited nodes, allowing for nodes to be processed in order based on their

The solution is implemented using Dijkstra's algorithm, a classical algorithm used in computing the shortest paths in a weighted

d, x, y = heappop(q)

2 ans = min(ans, d + dist(x, y, *target))

for x1, y1, x2, y2, cost in specialRoads:

heappush(q, (d + dist(x, y, x1, y1) + cost, x2, y2))

example that connects [1, 1] to [2, 2] with a cost of 1.

add the current position to the visited set.

of the special road to reduce costs.

1 from heapq import heappush, heappop

while priority_queue:

if (x, y) in visited:

continue

return minimum_cost

visited.add((x, y))

from typing import List

from math import inf

class Solution:

8

9

10

11

19

20

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

59

60

61

62

63

64

65

66

11 };

12

15

16

17

19

22

23

24

26

30

25 };

29 };

18 };

};

while (!frontier.empty()) {

if (visited.count(hashValue)) {

visited.insert(hashValue);

// Mark the current position as visited

for (auto& road : specialRoads) {

int roadCost = road[4];

// Return the calculated minimum cost

1 // Define comparator type for usage in the heap.

2 type Comparator<T> = (lhs: T, rhs: T) => number;

// Represents the priority queue using heap.

// Define the global heap structure with comparator.

// Calculate Manhattan distance between two points.

let heapComparator: (i: number, j: number) => boolean;

return Math.abs(x1 - x2) + Math.abs(y1 - y2);

const heapPush = (heap: any, element: any): void => {

return minimumCost;

let heapData: Array<any | null>;

heap.push(element);

let index = heapSize();

const top = heap.pop();

27 const heapSize = (): number => {

heapify(1);

return top;

const heapPop = (heap: any): any => {

return heapData.length - 1;

const heapify = (index: number): void => {

heapSwap(heap, 1, heapSize());

Typescript Solution

frontier.pop();

continue;

// Dequeue the closest (in terms of cost) position from the frontier

long long hashValue = static_cast<long long>(currentX) * hashMultiplier + currentY;

// Update minimum cost considering the current path and direct path to the target

// For each special road, calculate costs and push new positions to the queue

// Calculate cost to the start of the special road plus the road's cost

// Push the special road's end position with the updated cost into the queue

minimumCost = std::min(minimumCost, currentCost + calculateDistance(currentX, currentY, target[0], target[1]));

int newCost = currentCost + calculateDistance(currentX, currentY, startRoadX, startRoadY) + roadCost;

auto [currentCost, currentX, currentY] = frontier.top();

// Skip if this position has already been visited

int startRoadX = road[0], startRoadY = road[1];

int endRoadX = road[2], endRoadY = road[3];

frontier.push({newCost, endRoadX, endRoadY});

const calculateDistance = (x1: number, y1: number, x2: number, y2: number): number => {

while (index >> 1 !== 0 && heapComparator(index, index >> 1)) heapSwap(heap, index, (index >>= 1));

// Calculate the hash value of the current position

Now let's walk through how the algorithm works using Dijkstra's algorithm:

1 while q:

1 vis.add((x, y))

can be reached either directly (one step away in any of the four cardinal directions) or through a special road. The algorithm typically checks if the new path to a node is better than the previously known path, and updates it accordingly. Here, the dist function is defined to calculate the cost of moving between two positions on the grid. The priority queue q is initialized

- with a tuple containing the starting cost (0), and the x and y coordinates of the starting position. 1 def dist(x1: int, y1: int, x2: int, y2: int) -> int: return abs(x1 - x2) + abs(y1 - y2)
- q = [(0, start[0], start[1])] The solution maintains a loop iterating over the priority queue until it is empty, meaning all reachable positions have been considered.

Then it checks if the current position has been visited before: if (x, y) in vis: If not, the current position is added to the visited set, and we calculate the cost to reach the target from here:

The algorithm then considers all special roads, pushing the cost of traveling via each special road from the current node into the

priority queue. This cost includes the distance from the current node to the start of the special road plus the special road's cost:

Once the queue q is empty, the variable ans will contain the minimum cost to reach the target position, which is then returned.

Let's assume we start at position [0, 0] and aim to reach the target at position [2, 2]. The grid allows us to move at the cost of the

Manhattan distance between points, which is the sum of the absolute differences in the x and y coordinates. On top of the regular

movement, we have special roads that connect certain points at a potentially reduced cost. Let's consider one special road for our

Overall, the algorithm uses Dijkstra's pattern for pathfinding in an efficient manner, yielding the minimum cost considering both

Example Walkthrough

normal grid costs and special roads.

2. We also initialize a visited set vis to keep track of the positions we've already considered to avoid calculating them again.

3. We start by popping the first item from our priority queue: the starting position (0, 0, 0), with 0 being the cost so far.

4. Since [0, 0] isn't in our visited set, we process it. We calculate the Manhattan distance to our target [2, 2], which is 4, and we

5. Now, we consider the available special roads. In our case, there's one that goes from [1, 1] to [2, 2]. We calculate the cost to

get to the starting point of the special road, which is 2 (from [0, 0] to [1, 1]), and then add the cost of the special road (1).

1. We initialize our priority queue q with the starting position [(0, 0, 0)] which represents a 0 cost to reach [0, 0].

6. We put the endpoint of the special road [2, 2] into our priority queue with the calculated cost to get there (a total of 3). 7. Now the priority queue has one item, [3, 2, 2], which we then pop. Since this is the target and we haven't visited it before, we check the total cost required to reach here and update our answer ans if it's lower than any previous one.

8. As there are no other positions to process (the priority queue is empty, and there is only one special road in our example), the

In a more complex scenario with multiple special roads and a larger grid, the algorithm would work similarly, considering all possible

moves and special roads from each position visited, updating the priority queue and visited set accordingly until the minimum cost

algorithm ends. The minimum cost is 3, which is lower than the straight Manhattan distance to the target, demonstrating the use

to reach the target is found. Python Solution

def minimum_cost(self, start: List[int], target: List[int], special_roads: List[List[int]]) -> int:

Initialize a priority queue with a tuple containing distance, x coordinate, and y coordinate

Calculates Manhattan distance between two points (x1, y1) and (x2, y2)

Pop the node with the lowest distance from the priority queue

Compute total cost to reach the end of the special road

total_cost = distance + calculate_distance(x, y, x1, y1) + cost

Add the end point of the special road to the priority queue

minimum_cost = min(minimum_cost, distance + calculate_distance(x, y, *target))

def calculate_distance(x1: int, y1: int, x2: int, y2: int) -> int:

return abs(x1 - x2) + abs(y1 - y2)

distance, x, y = heappop(priority_queue)

Otherwise, add it to the visited set

for x1, y1, x2, y2, cost in special_roads:

// A large constant for the initial minimum cost

Set<Long> visited = new HashSet<>();

while (!priorityQueue.isEmpty()) {

if (visited.contains(hash)) {

for (int[] road : specialRoads) {

continue;

visited.add(hash);

int distance = point[0];

Check neighboring special roads

Return the minimum cost to reach the target

If the node has already been visited, skip it

Update the minimum cost if the new cost is lower

heappush(priority_queue, (total_cost, x2, y2))

public int minimumCost(int[] start, int[] target, int[][] specialRoads) {

// Set for visited points to avoid processing a point multiple times

// Add the starting point to the priority queue with initial cost 0

priorityQueue.offer(new int[] {0, start[0], start[1]});

// Process nodes until the priority queue is empty

int currentX = point[1], currentY = point[2];

// Unique number for the current point (hash)

// Skip if we've already visited this point

// Current distance from start to the point

// Explore special roads from the current point

long hash = 1L * currentX * gridSize + currentY;

int[] point = priorityQueue.poll();

12 priority_queue = [(0, start[0], start[1])] 13 # This set will keep track of visited points to prevent revisiting 14 visited = set() 15 # Initialize answer as infinity to track minimum cost 16 minimum_cost = inf 17 18 # Process nodes in the priority queue

// Method to calculate the minimum cost to travel from the start point to the target point considering special roads

// Update minimum cost with the sum of the current distance and direct distance from the current point to target

int roadStartX = road[0], roadStartY = road[1], roadEndX = road[2], roadEndY = road[3], roadCost = road[4];

minCost = Math.min(minCost, distance + calculateManhattanDistance(currentX, currentY, target[0], target[1]));

```
int minCost = Integer.MAX_VALUE;
          // Size of the grid
          int gridSize = 1000000;
8
          // Priority Queue to store states with minimum distance at the top
          PriorityQueue<int[]> priorityQueue = new PriorityQueue<>((a, b) -> a[0] - b[0]);
9
```

Java Solution

class Solution {

```
32
                     // Offer a new state to the queue with the updated cost considering the special road
 33
                     priorityQueue.offer(new int[] {
                         distance + calculateManhattanDistance(currentX, currentY, roadStartX, roadStartY) + roadCost,
 34
 35
                         roadEndX,
 36
                         roadEndY
                     });
 37
 38
 39
 40
             // Return the minimum cost found
             return minCost;
 41
 42
 43
 44
         // Helper method to calculate Manhattan distance between two points
 45
         private int calculateManhattanDistance(int x1, int y1, int x2, int y2) {
             return Math.abs(x1 - x2) + Math.abs(y1 - y2);
 46
 47
 48
 49
C++ Solution
  1 #include <vector>
  2 #include <queue>
    #include <unordered_set>
    #include <cmath>
    #include <tuple>
    class Solution {
    public:
         int minimumCost(vector<int>& start, vector<int>& target, vector<vector<int>>& specialRoads) {
             // Lambda function to calculate Manhattan distance between two points
 10
             auto calculateDistance = [](int x1, int y1, int x2, int y2) {
 11
 12
                 return std::abs(x1 - x2) + std::abs(y1 - y2);
 13
             };
 14
 15
             // Initialize the answer to a large number
             int minimumCost = INT MAX;
             // Define a very large value to be used as a hashing multiplier
 17
 18
             int hashMultiplier = 1e6;
             // Priority queue to maintain the frontier of the search
 19
 20
             std::priority_queue<std::tuple<int, int, int>,
 21
                                 std::vector<std::tuple<int, int, int>>,
 22
                                 std::greater<std::tuple<int, int, int>>> frontier;
 23
 24
             // Push the starting position with cost 0 into the priority queue
 25
             frontier.push({0, start[0], start[1]});
 26
             // Set to keep track of visited positions
             std::unordered_set<long long> visited;
 27
 28
```

33 34 35 36 37

while (true) { 32 let smallest = index; const left = index * 2; const right = index * 2 + 1; const size = heapData.length; if (left < size && heapComparator(left, smallest)) smallest = left;</pre> if (right < size && heapComparator(right, smallest)) smallest = right;</pre> 38 39 if (smallest !== index) { 40 heapSwap(heapData, index, smallest); 41 index = smallest; 42 } else break; 43 44 }; 45 const heapSwap = (heap: any, i: number, j: number): void => { 47 [heap[i], heap[j]] = [heap[j], heap[i]]; 48 }; 49 // Calculates the minimum cost to reach a target point from a start point, given special roads that can be used. const minimumCost = (start: number[], target: number[], specialRoads: number[][]): number => { 52 heapData = [null]; 53 heapComparator = (i, j) => heapData[i][0] < heapData[j][0]; 54 heapPush(heapData, [0, start[0], start[1]]); 55 56 const n = 1000000;57 const visited = new Set(); 58 let ans = Infinity; 59 60 while (heapSize()) { 61 const [distance, x, y] = heapPop(heapData); 62 const key = x * n + y; 63 if (visited.has(key)) { 64 continue; 65 66 visited.add(key); ans = Math.min(ans, distance + calculateDistance(x, y, target[0], target[1])); 67 68 for (const [x1, y1, x2, y2, cost] of specialRoads) { 69 heapPush(heapData, [distance + calculateDistance(x, y, x1, y1) + cost, x2, y2]); 70 71 72 73 74 return ans; 75 }; 76 // Set the comparator for the heap as a global definition which compares two elements based on their distance. const setComparator = (compareFunction: Comparator<any>): void => { heapComparator = (i, j) => compareFunction(heapData[i], heapData[j]) < 0; }; 80 81 // Initialize the heap with null at index 0 for easy index calculations. const initializeHeap = (): void => { heapData = [null]; 84 85 }; 86 // Initialize heap with custom comparator if provided. setComparator((lhs, rhs) => (lhs[0] < rhs[0] ? -1 : lhs[0] > rhs[0] ? 1 : 0)); 89 Time and Space Complexity

The time complexity of the given code is $0(n^2 \times n)$, where n is the number of special roads. This complexity arises because, in the worst case, each road is visited once and every time a road is visited, it could potentially lead to all other roads being considered as next steps. For each road, the distance to the next road is computed, which is an 0(1) operation, but then a point is added to the priority queue that has a size up to 0(n^2) (since every special road could be pushed to the queue with different starting points), and hence, the insertion time will be $0(\log n^2)$ which simplifies to $0(2(\log n) = 0(\log n)$. Since we could have $O(n^2)$ insertions, the total time complexity is $O(n^2 \times n)$.

worst case, each point in the grid could be added to the visited set and at most, every entry of the special roads could be stored in the priority queue simultaneously, if the points are all unique.

The space complexity of the code is $O(n^2)$ mainly due to the storage requirements of the priority queue and the visited set. In the