2662. Minimum Cost of a Path With Special Roads

Medium | Graph | Array | Shortest Path | Heap (Priority Queue) | 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., `|x2 - x1| + |y2 - y1|`.

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

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

Dijkstra's algorithm fits our needs well because it uses a priority queue to explore nodes (positions) that have the smallest known cost to reach from the start. It updates the cost to reach neighboring nodes as it progresses, and it's greedy in nature, always 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 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

The solution is implemented using Dijkstra's algorithm, a classical algorithm used in computing the shortest paths in a weighted graph. The specific data structures and patterns used in this implementation are:

- 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 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 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.

```python
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.

```python
while q:
    d, x, y = heappop(q)
```

Then it checks if the current position has been visited before:

```python
if (x, y) in vis:
    continue
```

If not, the current position is added to the visited set, and we calculate the cost to reach the target from here:

```python
vis.add((x, y))
ans = min(ans, d + dist(x, y, *target))
```

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:

```python
for x1, y1, x2, y2, cost in specialRoads:
    heappush(q, (d + dist(x, y, x1, y1) + cost, x2, y2))
```

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

Overall, the algorithm uses Dijkstra's pattern for pathfinding in an efficient manner, yielding the minimum cost considering both normal grid costs and special roads.

Example Walkthrough

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

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

1. We initialize our priority queue `q` with the starting position `[(0, 0, 0)]` which represents a `0` cost to reach `[0, 0]`.
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 add the current position to the `visited` set.
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`).
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 algorithm ends. The minimum cost is `3`, which is lower than the straight Manhattan distance to the target, demonstrating the use of the special road to reduce costs.

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 to reach the target is found.

Python Solution

```python
from heapq import heappush, heappop
from typing import List
from math import inf

class Solution:
    def minimum_cost(self, start: List[int], target: List[int], special_roads: List[List[int]]) -> int:
        # Calculates Manhattan distance between two points (x1, y1) and (x2, y2)
        def calculate_distance(x1: int, y1: int, x2: int, y2: int) -> int:
            return abs(x1 - x2) + abs(y1 - y2)

        # Initialize a priority queue with a tuple containing distance, x coordinate, and y coordinate
        priority_queue = [(0, start[0], start[1])]
        # This set will keep track of visited points to prevent revisiting
        visited = set()
        # Initialize answer as infinity to track minimum cost
        minimum_cost = inf

        # Process nodes in the priority queue
        while priority_queue:
            # Pop the node with the lowest distance from the priority queue
            distance, x, y = heappop(priority_queue)
            # If the node has already been visited, skip it
            if (x, y) in visited:
                continue
            # Otherwise, add it to the visited set
            visited.add((x, y))
            # Update the minimum cost if the new cost is lower
            minimum_cost = min(minimum_cost, distance + calculate_distance(x, y, *target))
            # Check neighboring special roads
            for x1, y1, x2, y2, cost in special_roads:
                # 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
                heappush(priority_queue, (total_cost, x2, y2))

        # Return the minimum cost to reach the target
        return minimum_cost
```

Java Solution

```java
class Solution {
    // Method to calculate the minimum cost to travel from the start point to the target point considering special roads
    public int minimumCost(int[] start, int[] target, int[][] specialRoads) {
        // A large constant for the initial minimum cost
        int minCost = Integer.MAX_VALUE;
        // Size of the grid
        int gridSize = 1000000;
        // Priority Queue to store states with minimum distance at the top
        PriorityQueue<int[]> priorityQueue = new PriorityQueue<>((a, b) -> a[0] - b[0]);
        // Set for visited points to avoid processing a point multiple times
        Set<Long> visited = new HashSet<>();
        // 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
        while (!priorityQueue.isEmpty()) {
            int[] point = priorityQueue.poll();
            int currentX = point[1], currentY = point[2];
            // Unique number for the current point (hash)
            long hash = 1L * currentX * gridSize + currentY;
            // Skip if we've already visited this point
            if (visited.contains(hash)) {
                continue;
            }
            visited.add(hash);
            // Current distance from start to the point
            int distance = point[0];
            // Update minimum cost with the sum of the current distance and direct distance from the current point to target
            minCost = Math.min(minCost, distance + calculateManhattanDistance(currentX, currentY, target[0], target[1]));
            // Explore special roads from the current point
            for (int[] road : specialRoads) {
                int roadStartX = road[0], roadStartY = road[1], roadEndX = road[2], roadEndY = road[3], roadCost = road[4];
                // Offer a new state to the queue with the updated cost considering the special road
                priorityQueue.offer(new int[] {
                    distance + calculateManhattanDistance(currentX, currentY, roadStartX, roadStartY) + roadCost,
                    roadEndX,
                    roadEndY
                });
            }
        }
        // Return the minimum cost found
        return minCost;
    }

    // Helper method to calculate Manhattan distance between two points
    private int calculateManhattanDistance(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);
    }
}
```

C++ Solution

```cpp
#include <vector>
#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
        auto calculateDistance = [](int x1, int y1, int x2, int y2) {
            return std::abs(x1 - x2) + std::abs(y1 - y2);
        };

        // Initialize the answer to a large number
        int minimumCost = INT_MAX;
        // Define a very large value to be used as a hashing multiplier
        int hashMultiplier = 1e6;
        // Priority queue to maintain the frontier of the search
        std::priority_queue<std::tuple<int, int, int>,
                            std::vector<std::tuple<int, int, int>>,
                            std::greater<std::tuple<int, int, int>>> frontier;

        // Push the starting position with cost 0 into the priority queue
        frontier.push({0, start[0], start[1]});
        // Set to keep track of visited positions
        std::unordered_set<long long> visited;

        while (!frontier.empty()) {
            // Dequeue the closest (in terms of cost) position from the frontier
            auto [currentCost, currentX, currentY] = frontier.top();
            frontier.pop();

            // Calculate the hash value of the current position
            long long hashValue = static_cast<long long>(currentX) * hashMultiplier + currentY;

            if (visited.count(hashValue)) {
                // Skip if this position has already been visited
                continue;
            }

            // Mark the current position as visited
            visited.insert(hashValue);

            // Update minimum cost considering the current path and direct path to the target
            minimumCost = std::min(minimumCost, currentCost + calculateDistance(currentX, currentY, target[0], target[1]));

            // For each special road, calculate costs and push new positions to the queue
            for (auto& road : specialRoads) {
                int startRoadX = road[0], startRoadY = road[1];
                int endRoadX = road[2], endRoadY = road[3];
                int roadCost = road[4];

                // Calculate cost to the start of the special road plus the road's cost
                int newCost = currentCost + calculateDistance(currentX, currentY, startRoadX, startRoadY) + roadCost;

                // Push the special road's end position with the updated cost into the queue
                frontier.push({newCost, endRoadX, endRoadY});
            }
        }

        // Return the calculated minimum cost
        return minimumCost;
    }
};
```

Typescript Solution

```typescript
// Define comparator type for usage in the heap.
type Comparator<T> = (lhs: T, rhs: T) => number;

// Define the global heap structure with comparator.
let heapData: Array<any | null>;
let heapComparator: (i: number, j: number) => boolean;

// Calculate Manhattan distance between two points.
const calculateDistance = (x1: number, y1: number, x2: number, y2: number): number => {
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
};

// Represents the priority queue using heap.
const heapPush = (heap: any, element: any): void => {
    heap.push(element);
    let index = heapSize();
    while (index >> 1 !== 0 && heapComparator(index, index >> 1)) heapSwap(heap, index, (index >>= 1));
};

const heapPop = (heap: any): any => {
    heapSwap(heap, 1, heapSize());
    const top = heap.pop();
    heapify(1);
    return top;
};

const heapSize = (): number => {
    return heapData.length - 1;
};

const heapify = (index: number): void => {
    while (true) {
        let smallest = index;
        const left = index * 2;
        const right = index * 2 + 1;
        const size = heapData.length;
        if (left < size && heapComparator(left, smallest)) smallest = left;
        if (right < size && heapComparator(right, smallest)) smallest = right;
        if (smallest !== index) {
            heapSwap(heapData, index, smallest);
            index = smallest;
        } else break;
    }
};

const heapSwap = (heap: any, i: number, j: number): void => {
    [heap[i], heap[j]] = [heap[j], heap[i]];
};

// 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 => {
    heapData = [null];
    heapComparator = (i, j) => heapData[i][0] < heapData[j][0];
    heapPush(heapData, [0, start[0], start[1]]);

    const n = 1000000;
    const visited = new Set();
    let ans = Infinity;

    while (heapSize()) {
        const [distance, x, y] = heapPop(heapData);
        const key = x * n + y;
        if (visited.has(key)) {
            continue;
        }
        visited.add(key);
        ans = Math.min(ans, distance + calculateDistance(x, y, target[0], target[1]));

        for (const [x1, y1, x2, y2, cost] of specialRoads) {
            heapPush(heapData, [distance + calculateDistance(x, y, x1, y1) + cost, x2, y2]);
        }
    }

    return ans;
};

// 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;
};

// Initialize the heap with null at index 0 for easy index calculations.
const initializeHeap = (): void => {
    heapData = [null];
};

// Initialize heap with custom comparator if provided.
setComparator((lhs, rhs) => (lhs[0] < rhs[0] ? -1 : lhs[0] > rhs[0] ? 1 : 0));
```

Time and Space Complexity

The time complexity of the given code is $O(n^2 \times \log n)$, 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 $O(1)$ operation, but then a point is added to the priority queue that has a size up to $O(n^2)$ (since every special road could be pushed to the queue with different starting points), and hence, the insertion time will be $O(\log n^2)$ which simplifies to $O(2\log n) = O(\log n)$. Since we could have $O(n^2)$ insertions, the total time complexity is $O(n^2 \times \log n)$.

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 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.