1368. Minimum Cost to Make at Least One Valid Path in a Grid **Breadth-First Search** Graph Array Matrix **Shortest Path** Heap (Priority Queue) **Leetcode Link** Hard

Problem Description

0) and end at the bottom right cell (m - 1, n - 1). Each cell in the grid contains a sign which directs you to the next cell to visit. The • 1: Move right to the cell (i, j + 1)

The task is to find the minimum cost to establish at least one valid path in an m x n grid. This path must start at the top left cell (0,

- possible signs are:
- 3: Move down to the cell (i + 1, j) • 4: Move up to the cell (i - 1, j)

• 2: Move left to the cell (i, j - 1)

- Some signs might point outside of the grid boundaries, which are to be considered invalid directions for the purpose of a valid path.
- bottom right cell exists.

The objective is to determine the minimum cost to alter the signs in such a way that at least one valid path from the top left to the

Intuition To solve this problem, we use a breadth-first search (BFS) approach, but with a slight tweak. This problem can be thought of as

traversing a graph where each cell represents a node, and the signs are the directed edges to the neighboring nodes. The challenge,

however, lies in the fact that these directed edges can be altered at a cost.

is what we return.

Solution Approach

The modified BFS algorithm uses a double-ended queue (deque) to keep track of the cells to be visited. This is crucial as the queue can have elements added to both its front and back, which helps in maintaining the order of traversal based on the cost associated with moving to a particular cell. Here's the intuition behind the BFS traversal:

1. We begin at the starting cell (0, 0) with an initial cost of 0. 2. As we visit a cell, we look at all possible directions we could move from that cell. 3. If the direction aligns with the arrow currently in the cell, we can move to that cell at no additional cost. In this case, we add this cell at the front of the deque to prioritize it.

4. If the direction doesn't align with the arrow, we would need to change the sign with a cost of 1. For these cells, we add them to

the back of the deque as they represent potential paths but at a higher cost. 5. Each cell is visited only once to ensure minimal traversal cost, thus, we maintain a visited set.

6. This process continues until we reach the destination cell (m - 1, n - 1) or there are no more cells left to visit in the deque.

1. Initialize the dimensions m (rows) and n (columns) of the grid.

7. The cost associated with the first visit to the destination cell is the minimum cost needed to make at least one valid path, which

The cost to change a sign in any cell is 1, and each sign can be changed only once.

- By always choosing to traverse in the indicated direction without any cost, we ensure that we are taking advantage of the free
- moves as much as possible before incurring any additional costs. The visited set prevents us from revisiting cells and possibly entering a loop.
- The provided reference code implements a BFS strategy using a deque. Here's a step-by-step breakdown of the solution's implementation:

2. Define an array dirs to represent the possible directions of travel based on the grid's signs: [[0, 0], [0, 1], [0, -1], [1, 0], [-1, 0]]. Each sub-array corresponds to the deltas for the row and column indices when moving in each of the four directions. 3. Create a double-ended queue q, initialized with a tuple containing the row index 0, column index 0, and the initial cost 0. 4. Create a set vis to keep track of visited cells and prevent revisiting them.

5. Start a loop to continue processing until the deque q is empty. Each iteration will handle one cell's visit:

• Dequeue an element (i, j, d) from the front of q, where i and j are the current cell's indices, and d is the cost to reach this cell.

current indices i and j.

∘ If the cell (i, j) has already been visited, skip processing it to avoid loops and unnecessary cost increments. Mark the current cell (i, j) as visited by adding it to vis. \circ If the cell (i, j) is the destination (m-1, m-1), return the cost d because a valid path has been found.

• For each possible direction k (1 to 4), calculate the indices of the adjacent cell (x, y) by adding the directional deltas to the

- 6. Check if the new cell (x, y) is within the bounds of the grid: o If the sign at the current cell grid[i][j] indicates the direction k, meaning no sign change is needed, add (x, y, d) to the front of q, not increasing the cost, as it's a free move in the desired direction.
- ∘ If the sign does not match, add (x, y, d + 1) to the back of q, increasing the cost by 1 to account for the change of sign. 7. If no valid path is found by the end of the traversal, return -1. The use of a deque allows for efficient addition of cells to either the front or back, depending on whether a cost is incurred. By

vis prevents revisiting and recalculating paths for positions that have already been evaluated, which optimizes the search. This

prioritizing the moves that don't require sign changes (zero cost), the algorithm ensures the minimum cost path is found first. The set

approach effectively treats the grid as a graph and considers the costs of edges dynamically, resulting in a clever application of the

BFS algorithm adapted for weighted pathfinding.

Let's assume we have a 3×3 grid, where the signs in the cells are arranged as follows:

1. We start at the top-left cell (0, 0) with an initial cost of 0. 2. From here, the sign 1 tells us to move right to cell (0, 1). This is a valid move with no cost so we add (0, 1) to the front of our deque. 3. Now we consider the cell (0, 1) with the sign 3, which tells us to move down. The target cell (1, 1) has not been visited, so we add (1, 1) to the front of the deque again without any additional cost.

4. At cell (1, 1), the sign 2 indicates moving left to (1, 0). This cell has not been visited, so we add (1, 0) to the front of the

5. Next, we visit cell (1, 0) which contains the sign 4, telling us to move up to (0, 0). However, this has been visited, so we don't

8. Now, we get to cell (1, 0) and from here, the direction 4 (up) is not valid since it would take us outside the grid or into visited

9. Cell (1, 1) won't be processed as it's already visited, so we look at cell (2, 0) with the sign 3 which leads us down to (2, 1).

11. Finally, we arrive at the bottom right cell (2, 2) and our destination is reached without needing further sign changes. The cost

cells. At this point, we need to consider changing the direction, so we will add the neighboring cells (1, 1) to the right and (2,

7. Continuing this process, we encounter cell (1, 1) again, which we've visited, so we move on.

add anything to our deque.

from collections import deque

class Solution:

5

6

8

9

10

11

12

13

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

44

45

46

47

48

deque.

Example Walkthrough

1 1 3 4

There's no cost for moving down since the sign matches. We add (2, 1) to the front of the deque. 10. From cell (2, 1), we move right to (2, 2) as indicated by the 1 sign. As this is a free move, it is added to the front of the deque.

0) down to the back of the deque with an added cost of 1.

6. Our next cell would be (0, 1) again, but since it's visited, we ignore it.

Now let's walk through the solution approach with this grid:

at this point is 1. The minimum cost required to establish at least one valid path in this grid is 1, which entails changing one sign. By strategically using

def minCost(self, grid: List[List[int]]) -> int:

rows, cols = len(grid), len(grid[0])

i, j, cost = queue.popleft()

Mark the current cell as visited

if i == rows - 1 and j == cols - 1:

if grid[i][j] == k:

Check for valid cell coordinates

if (i, j) in visited:

continue

visited.add((i, j))

return cost

for k in range(1, 5):

else:

queue = deque([(0, 0, 0)])

Initialize rows and cols with the dimensions of the grid

directions = [[0, 0], [0, 1], [0, -1], [1, 0], [-1, 0]]

Define the direction vectors for right, left, down, up respectively

Initialize queue with the starting point and the current cost (0)

Check if the cell is already visited to avoid redundancy

Explore all possible directions from the current cell

queue.appendleft((x, y, cost))

queue.append((x, y, cost + 1))

If no path found to the bottom right corner return -1

x, y = i + directions[k][0], j + directions[k][1]

Check if we have reached the bottom right corner, return cost if true

- a deque and a visited set, we conducted a breadth-first search that prioritized no-cost moves over those requiring a sign change. Python Solution
- 14 # Set to maintain visited cells 15 visited = set() 16 17 # Iterate until the queue is empty while queue: 18 # Pop the cell from queue with its current cost 19

If the current direction matches the arrow in the grid cell, no cost is added

If the direction is different, add a cost of 1 to switch the arrow

if 0 <= x < rows and 0 <= y < cols:</pre> 39 40 41 42 43

return -1

```
Java Solution
     class Solution {
         public int minCost(int[][] grid) {
             // m holds the number of rows in the grid.
             int numRows = grid.length;
             // n holds the number of columns in the grid.
             int numCols = grid[0].length;
             // vis holds information whether a cell has been visited.
  8
             boolean[][] visited = new boolean[numRows][numCols];
             // Queue for performing BFS with modifications for 0-cost moves.
  9
             Deque<int[]> queue = new ArrayDeque<>();
 10
             // Starting by adding the top-left cell with 0 cost.
 11
 12
             queue.offer(new int[] {0, 0, 0});
             // dirs are used to navigate throughout the grid. (right, left, down, up)
 13
 14
             int[][] directions = {{0, 0}, {0, 1}, {0, -1}, {1, 0}, {-1, 0}};
 15
 16
             // BFS starts here
 17
             while (!queue.isEmpty()) {
 18
                 // Dequeue a cell info from the queue.
 19
                 int[] position = queue.poll();
 20
                 // i and j hold the current cell row and column, d holds the current cost.
 21
                 int i = position[0], j = position[1], cost = position[2];
 22
 23
                 // If we've reached the bottom-right cell, return the cost.
                 if (i == numRows - 1 && j == numCols - 1) {
 24
 25
                      return cost;
 26
 27
                 // If this cell is already visited, skip it.
 28
                 if (visited[i][j]) {
 29
                     continue;
 30
 31
                 // Mark the cell as visited.
 32
                 visited[i][j] = true;
 33
 34
                 // Explore all possible directions from the current cell.
 35
                 for (int k = 1; k \le 4; ++k) {
 36
                      int newX = i + directions[k][0], newY = j + directions[k][1];
                     // Check the validity of the new cell coordinates.
 37
 38
                      if (\text{newX} >= 0 \& \& \text{newX} < \text{numRows} \& \& \text{newY} >= 0 \& \& \text{newY} < \text{numCols}) 
 39
                          // If the current direction is the same as the arrow in this cell (no cost to move here).
 40
                         if (grid[i][j] == k) {
 41
                              // Add the new cell at the front of the queue to explore it sooner (as it's no cost).
 42
                              queue.offerFirst(new int[] {newX, newY, cost});
 43
                          } else {
 44
                              // Otherwise, add the new cell at the end of the queue and increase the cost by 1.
 45
                              queue.offer(new int[] {newX, newY, cost + 1});
 46
 47
 48
 49
 50
             // If the queue is empty and we didn't reach the bottom-right cell, return -1 as it's not possible.
```

32 33 34 35

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

57

1;

51

52

53

54

C++ Solution

2 public:

6

8

12

13

1 class Solution {

return -1;

int minCost(vector<vector<int>>& grid) {

int rows = grid.size(), cols = grid[0].size();

// Initialize a 2D vector to keep track of visited cells

vector<vector<bool>> visited(rows, vector<bool>(cols, false));

// Define directions according to the grid value (1: right, 2: left, 3: down, 4: up)

vector<vector< $int>> directions = {\{0, 0\}, \{0, 1\}, \{0, -1\}, \{1, 0\}, \{-1, 0\}\};}$

// Queue to perform the BFS, holding pairs of (cell_index, current_cost)

// Get dimensions of the grid

```
14
             deque<pair<int, int>> queue;
 15
             queue.push_back({0, 0}); // Start from the top-left corner with 0 cost
 16
 17
             // While there are elements in the queue
             while (!queue.empty()) {
 18
 19
                 // Get the front element
 20
                 auto current = queue.front();
 21
                 queue.pop_front();
 22
 23
                 // Calculate row and column from the cell index
 24
                 int row = current.first / cols, col = current.first % cols;
 25
                 int cost = current.second;
 26
 27
                 // If we've reached the bottom-right corner, return the current cost
                 if (row == rows - 1 && col == cols - 1) return cost;
 28
 29
 30
                 // If the current cell is already visited, skip it
 31
                 if (visited[row][col]) continue;
                 // Mark the current cell as visited
                 visited[row][col] = true;
 36
                 // Check all four adjacent cells
 37
                 for (int k = 1; k \le 4; ++k) {
 38
                     int newRow = row + directions[k][0], newCol = col + directions[k][1];
 39
                     // If the new cell is within the grid bounds
 40
                     if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {</pre>
 41
 42
                         // Calculate the new cell index
 43
                         int newIndex = newRow * cols + newCol;
 44
 45
                         // If the grid indicates the current direction, add to the front of the queue without increasing cost
                         if (grid[row][col] == k)
 46
 47
                             queue.push_front({newIndex, cost});
                         // Otherwise, the direction is not towards the grid arrow — increase the cost and add to the back of the queue
 48
 49
                         else
 50
                             queue.push_back({newIndex, cost + 1});
 51
 52
 53
 54
 55
             // Return -1 if the bottom-right corner could not be reached
 56
             return -1;
 57
 58
    };
 59
Typescript Solution
    function minCost(grid: number[][]): number {
         // Dimensions of the grid.
         const rows = grid.length,
             cols = grid[0].length;
  6
         // Initialize the answer grid with infinity, indicating no paths have been taken.
         let costGrid = Array.from({ length: rows }, () => new Array(cols).fill(Infinity));
  8
         costGrid[0][0] = 0; // Starting point cost is zero.
  9
 10
         // Queue for BFS traversal, starting from the top-left cell.
 11
         let queue: [number, number][] = [[0, 0]];
 12
 13
         // Directions for right, left, down, up.
 14
         const directions = [
 15
             [0, 1], // right
 16
             [0, -1], // left
 17
             [1, 0], // down
```

52 53 54 // Return the minimum cost to reach the bottom right corner. 55 return costGrid[rows - 1][cols - 1]; 56 }

Time and Space Complexity

[-1, 0], // up

while (queue.length) {

// Perform BFS on the grid.

// Current position.

let [x, y] = queue.shift()!;

// Explore all possible directions.

// Get the next direction.

// Compute the new position.

let [i, j] = [x + dx, y + dy];

// Check boundary conditions.

for (let step = 1; step < 5; step++) {</pre>

let [dx, dy] = directions[step - 1];

if (cost >= costGrid[i][j]) continue;

// Update the cost at the new position.

if (i < 0 || i >= rows || j < 0 || j >= cols) continue;

// Skip if the new cost is not less than the current cost at position (i, j).

// Calculate the cost to enter the new position.

let cost = ~~(grid[x][y] != step) + costGrid[x][y];

being processed sequentially. Hence, the overall space complexity remains 0(m * n).

43 costGrid[i][j] = cost; 44 45 // Prioritize movement in the direction the arrow points by placing it at the front of the queue. if (grid[x][y] == step) { 46 queue.unshift([i, j]); 47 48 } else { // Otherwise, enqueue normally. 49 queue.push([i, j]); 50 51

Time Complexity:

the number of cells. **Space Complexity:** The space complexity is also 0(m * n) because of the visited set (vis) storing up to m * n unique cell positions to ensure cells are

The provided code solves the problem using a Breadth First Search (BFS) algorithm with a slight optimization. The algorithm has two

modes of queue operations: normally numbers are appended to the end of the queue, but when moving in the direction the grid

The time complexity is 0(m * n), where m is the number of rows and n is the number of columns in the grid. This is because in the

iterate over directions, they only add constant work at each node, so it does not affect the overall linear complexity with respect to

worst-case scenario, each cell is visited only once due to the use of the visited (vis) set. Even though there is a nested loop to

arrow points to, it's added to the front. This has the potential to reduce the number of steps needed to reach the end.

not revisited. Additionally, the queue (q) in the worst case may also contain elements from all the cells in the grid when they are