Minimum Time To Visit A Cell In A Grid

solved by	Senan
Platform	LeetCode
difficulty	Hard
# Serial	2577
tags	Dijkstras Algorithm
language	C++
solved on	@29/11/2024
link	https://leetcode.com/problems/minimum-time-to-visit-a-cell-in-a-grid/description/
Completion	☑

Intuition

The problem involves navigating a grid where each cell has a time constraint. The goal is to determine the minimum time required to reach the bottom-right cell `(m-1, n-1)` from the top-left cell `(0, 0)`. The challenge lies in navigating efficiently while respecting the time constraints at each grid cell.

The intuition is similar to Dijkstra's algorithm for finding the shortest path in a weighted graph, but here, the weight depends on time constraints.

Approach

1. **Priority Queue (Min-Heap):** Use a priority queue to always process the cell with the current minimum time.
2. **Distance Matrix:** Maintain a `dist` matrix to store the minimum time to reach each cell. Initialize all cells to `INT_MAX`.
3. **Four Directions:** Use `dr` and `dc` arrays to represent the four possible moves (up, down, left, right).
4. **Cell Processing:**
   - Pop the cell with the smallest time from the priority queue.
   - Check if it is the destination `(m-1, n-1)`. If yes, return the current time.
   - For each neighbor, calculate the new time `nTime` considering the constraints:
     - If `(grid[nRow][nCol] - time)` is even, add 1 to align with the constraint.
     - Update the time only if the calculated time is less than the recorded time in `dist`.
   - Push the updated neighbor state back into the priority queue.

Complexity

Time Complexity:

- Let `m` be the number of rows and `n` be the number of columns.

- The priority queue processes each cell once, and each insertion or deletion takes `O(log(m * n))`.
- Overall complexity:**O(m * n * log(m * n))**

Space Complexity:

- The `dist` matrix takes `O(m * n)` space.
- The priority queue holds at most `m * n` elements.
- Overall space complexity:**O(m * n)**

Code

```
class Solution {
public:
    int minimumTime(vector<vector<int>>& grid) {
        if (grid[1][0] > 1 && grid[0][1] > 1)  return -1;
        int m = grid.size(), n = grid[0].size();

        vector<int> dr = {-1, 0, 1, 0};
        vector<int> dc = {0, -1, 0, 1};

        vector<vector<int>> dist(m, vector<int>(n, INT_MAX));

        priority_queue<vector<int>, vector<vector<int>>,
                       greater<vector<int>>>  pq;

        pq.push({0, 0, 0});

        while (!pq.empty()) {
            int time = pq.top()[0];
            int row = pq.top()[1];
            int col = pq.top()[2];
            pq.pop();

            if(row == m-1 && col == n-1) return time;

            for (int i = 0; i < 4; i++){
                int nRow = row + dr[i];
                int nCol = col + dc[i];
                if(0 <= nRow && nRow < m && 0 <= nCol && nCol < n){
                    int diff = !((grid[nRow][nCol] - time) & 1);
                    int nTime = max(time + 1, grid[nRow][nCol] + diff);

                    if(nTime < dist[nRow][nCol]){
                        dist[nRow][nCol] = nTime;
                        pq.push({nTime, nRow, nCol});
                    }
                }
            }
        }

        return -1;
    }
};
```