286. Walls and Gates (/problems/walls-and-gates/)

March 5, 2016 | 6K views

You are given a *m x n* 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value $2^{31} - 1 = 2147483647$ to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with `INF`.

For example, given the 2D grid:

```
INF  -1  0  INF
INF INF INF  -1
INF  -1 INF  -1
  0  -1 INF INF
```

After running your function, the 2D grid should be:

```
  3  -1   0   1
  2   2   1  -1
  1  -1   2  -1
  0  -1   3   4
```

Solution

Approach #1 (Brute Force) [Time Limit Exceeded]

The brute force approach is simple, we just implement a breadth-first search from each empty room to its nearest gate.

While we are doing the search, we use a 2D array called `distance` to keep track of the distance from the starting point. It also implicitly tell us whether a position had been visited so it won't be inserted into the queue again.

```
private static final int EMPTY = Integer.MAX_VALUE;
private static final int GATE = 0;
private static final int WALL = -1;
private static final List<int[]> DIRECTIONS = Arrays.asList(
        new int[] { 1,  0},
        new int[] {-1,  0},
        new int[] { 0,  1},
        new int[] { 0, -1}
);

public void wallsAndGates(int[][] rooms) {
    if (rooms.length == 0) return;
    for (int row = 0; row < rooms.length; row++) {
        for (int col = 0; col < rooms[0].length; col++) {
            if (rooms[row][col] == EMPTY) {
                rooms[row][col] = distanceToNearestGate(rooms, row, col);
            }
        }
    }
}

private int distanceToNearestGate(int[][] rooms, int startRow, int startCol) {
    int m = rooms.length;
    int n = rooms[0].length;
    int[][] distance = new int[m][n];
    Queue<int[]> q = new LinkedList<>();
    q.add(new int[] { startRow, startCol });
    while (!q.isEmpty()) {
        int[] point = q.poll();
        int row = point[0];
        int col = point[1];
        for (int[] direction : DIRECTIONS) {
            int r = row + direction[0];
            int c = col + direction[1];
            if (r < 0 || c < 0 || r >= m || c >= n || rooms[r][c] == WALL
                    || distance[r][c] != 0) {
                continue;
            }
            distance[r][c] = distance[row][col] + 1;
            if (rooms[r][c] == GATE) {
                return distance[r][c];
            }
            q.add(new int[] { r, c });
        }
    }
    return Integer.MAX_VALUE;
}
```

Complexity analysis

- Time complexity : $O(m^2n^2)$. For each point in the $m \times n$ size grid, the gate could be at most $m \times n$ steps away.
- Space complexity : $O(mn)$. The space complexity depends on the queue's size. Since we won't insert points that have been visited before into the queue, we insert at most $m \times n$ points into the queue.

Approach #2 (Breadth-first Search) [Accepted]

Instead of searching from an empty room to the gates, how about searching the other way round? In other words, we initiate breadth-first search (BFS) from all gates at the same time. Since BFS guarantees that we search all rooms of distance *d* before searching rooms of distance *d* + 1, the distance to an empty room must be the shortest.

```
private static final int EMPTY = Integer.MAX_VALUE;
private static final int GATE = 0;
private static final List<int[]> DIRECTIONS = Arrays.asList(
        new int[] { 1,  0},
        new int[] {-1,  0},
        new int[] { 0,  1},
        new int[] { 0, -1}
);

public void wallsAndGates(int[][] rooms) {
    int m = rooms.length;
    if (m == 0) return;
    int n = rooms[0].length;
    Queue<int[]> q = new LinkedList<>();
    for (int row = 0; row < m; row++) {
        for (int col = 0; col < n; col++) {
            if (rooms[row][col] == GATE) {
                q.add(new int[] { row, col });
            }
        }
    }
    while (!q.isEmpty()) {
        int[] point = q.poll();
        int row = point[0];
        int col = point[1];
        for (int[] direction : DIRECTIONS) {
            int r = row + direction[0];
            int c = col + direction[1];
            if (r < 0 || c < 0 || r >= m || c >= n || rooms[r][c] != EMPTY) {
                continue;
            }
            rooms[r][c] = rooms[row][col] + 1;
            q.add(new int[] { r, c });
        }
    }
}
```

Complexity analysis

- Time complexity : $O(mn)$.

  If you are having difficulty to derive the time complexity, start simple.

  Let us start with the case with only one gate. The breadth-first search takes at most $m \times n$ steps to reach all rooms, therefore the time complexity is $O(mn)$. But what if you are doing breadth-first search from k gates?

  Once we set a room's distance, we are basically marking it as visited, which means each room is visited at most once. Therefore, the time complexity does not depend on the number of gates and is $O(mn)$.

- Space complexity : $O(mn)$. The space complexity depends on the queue's size. We insert at most $m \times n$ points into the queue.

G (https://discuss.leetcode.com/user/gregpen)

gregpen commented 4 months ago

If a room can't be reached from any of the gates, it will stay filled with INF, so the solution still works. If some other entry was requested for unreached rooms it would be an easy task to rewalk the grid again after the BFS gate walk and mark remaining, unreached rooms.

A (https://discuss.leetcode.com/user/ark-kun)

Ark-kun commented 4 months ago

> "If it is impossible to reach a gate, it should be filled with INF."
> Reach from where? The description is ambiguous. My solution was correct, but only by chance.

G (https://discuss.leetcode.com/user/gregpen)

gregpen commented 4 months ago

I think what some folks are missing in this second solution is that each gate is not fully searched before moving on to a new gate. Each gate only looks at the areas within 1 space before we check the next gate. So each area within one space of the gates are checked for rooms and these rooms are marked, then added to the queue. Once all gates are checked, each new space is checked, and so forth. So, once a room gets hit, it has to be from the closest gate.

 (https://discuss.leetcode.com/user/flagbigoffer)

FLAGbigoffer commented 7 months ago

Good job! The second solution is beautiful and definitely right. Because we use BFS in each GATE, each time we move one step and mark the shortest in one cell. As long as this cell is marked, we don't have to visit and re-mark it again. That's it. Scan the whole matrix one time. Time complexity definitely O(m * n).

 (https://discuss.leetcode.com/user/huimengpaopao)

huimengpaopao commented last year

This answer is wrong.

 (https://discuss.leetcode.com/user/jdrogin)

jdrogin commented last year

I think it is still O(m * n) because
step 1: you iterate over all rows and columns giving m*n operations each of O(1).*
*step 2: then you start at each gate and do 4 directions, adding in other cells until you have looked at every cell. The key here is that you will only look at each cell once during this process, so again you have O(m*n) it doesn't matter how many initial gates you have you will still iterate over every cell once during this process.

The result of these 2 steps is O(m*n). Number of gates doesn't matter because you are marking the cells completed as your process them to ensure you process each cell once regardless of what was in your initial queue of starting points.

L (https://discuss.leetcode.com/user/liu971)

liu971 commented last year

Hi, isn't the time complexity O(m * n * k) where k is the number of gates.


(https://discuss.leetcode.com/user/jdrogin)

jdrogin commented last year

Maybe this will help you to see how it works:
Add all the Gates (zeros) to the Queue, this is the first step, done prior to starting the BFS loop.
Now you start BFS from each of the found Gates, expanding outward. When you encounter a cell that has not been reached yet (you know this because it's value is INF) it will necessarily be one more than your current cell. If you reached it quicker via some other Gate it's value would have already been set (no longer INF) and thus your path is longer, so ignore it. As you find cells that you have not reached yet, after setting their value, add them to the Queue. Eventually the queue is empty and all the reachable cells have been treated.

P
(https://discuss.leetcode.com/user/piyush121)

piyush121 commented last year

It will still work @ichuen (https://discuss.leetcode.com/uid/52490) because the BFS algorithm will make sure that when the empty room is discovered first, that means the distance is the shortest at that time.

View original thread (https://discuss.leetcode.com/topic/40)

Load more comments...