**663. Walls and Gates**

<https://www.lintcode.com/problem/walls-and-gates/description>

1. **Listen**

**Problem Statement:**

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, that room should remain filled with INF

**Input:**

You are given an **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.

**Goal:**

Fill each empty room with the distance to its nearest gate.

If it is impossible to reach a Gate, that room should remain filled with INF.

**Return:**

return the grid edited in-place.

all the empty rooms should be filled with the distance to the nearest gate

1. **Examples**

**Example 1:**

**A picture containing text, crossword puzzle

Description automatically generated**

**Input:**

[[INF, -1, 0, INF],

[INF, INF, INF ,-1],

[INF, -1, INF, -1],

[0, -1, INF, INF]]

**Output:**

[[3,-1,0,1],

[2,2,1,-1],

[1,-1,2,-1],

[0,-1,3,4]]

**Explanation:**

the 2D grid is:

INF -1 0 INF

INF INF INF -1

INF -1 INF -1

0 -1 INF INF

the answer is:

3 -1 0 1

2 2 1 -1

1 -1 2 -1

0 -1 3 4

**Example 2:**

**Input:**

[[0,-1],[2147483647,2147483647]]

**Output:**

[[0,-1],[1,2]]

**Constraints:**

* **N/A**

**Test Cases:**

1. **Brute Force**

**Solution 1: Time = O((MN)^2), Space = ((M+N)^2)**

* We scan over the matrix running a **DFS** on **every empty** **room cell**.
* The DFS function will return what the distance to the closest gate from the current cell.
* Note that DFS will not change any cell values.
* Traversing over a matrix takes MN time, and each DFS would also take up to MN time.
* Not optimal solution because there is a lot of repeated work.

**Solution 2:**

* We can use a **BFS** shortest path solution on **every empty** **room cell**.
* The BFS function returns the shortest distance to the nearest gate.
* Note that BFS will not change any cells values.
* This has the same pitfall as the DFS solution, in that there will be a lot of repeated work.

**Solution 3:**

* Let’s reverse our thinking.
* We can use a **BFS** shortest path solution from **every gate**.
* The BFS function iterates over all **adjacent** empty room cells and replaces their current value with the distance from the gate.

**Diagram

Description automatically generated**

* However, we are still repeating work, except at the expense of a smaller constant factor.
* We compute BFS on all gates instead of all empty rooms.

1. **Optimize**

**Solution 4:**

* An even better solution would be to run a **BFS** solution from **every gate simultaneously**.
* We first initialize our queue with all gates in the matrix.
* The BFS function iterates over all **adjacent** empty room cells from every gate **at the same time** and replaces their current value with the distance from the gate.
  + Iteration 1 – Replace all unvisited adjacent empty rooms 1 distance away from all gates with a value of 1. Add them to the queue
  + Iteration 2 – Replace all unvisited adjacent empty rooms 2 distance away from all gates with a value of 2. Add them to the queue
  + Iteration 3 – Replace all unvisited adjacent empty rooms 3 distance away from all gates with a value of 3. Add them to the queue

…

stop when our queue is empty

* Which rooms are unvisited?
  + We know if a cell has been visited before if it does not have an INF.
  + We also know the only cells we even have to process cannot be -1 or 0.
  + The only cells we have to process on each iteration are adjacent cells that have an INF value.
* Ensuring we only visit unvisited cells means that all cells are visited once, and that all cells have their proper distance value, as seen in the following example:

Diagram

Description automatically generated

The runtime of this algorithm is O(MN). We load the queue in O(MN) time and traverse all cells in the matrix once using BFS in O(MN) time. Therefore, the runtime is O(MN + MN) = O(MN)

The space complexity is O(MN), because we could receive a matrix entirely filled with gates.

1. **Walkthrough**

We iterate over the matrix and add any gates we see to a queue.

This queue stores the index of the gate.

We keep a distance variable that is incremented for each BFS traversal we perform.

While the queue is not empty

for each cell in the queue

get current cell indices from the queue

for all 4 adjacent neighbors of cell

if cell has not been visited and is empty room

replace cell value with current cell + 1

add to queue

1. **Implement**

public void wallsAndGates(int[][] rooms) {

int mRows = rooms.length, nCols = rooms[0].length;

Queue<int[]> queue = new Queue<>();

for(int i = 0; i < mRows; i++) {

for(int j = 0; j < nCols; j++) {

if(rooms[i][j] == 0) queue.add(new int[] {i, j});

}

}

int distance = 0;

int[] dirs = new int[] {{0,1}, {1,0}, {0,-1}, {-1,0}};

while(!queue.isEmpty()) {

distance++;

int[] cell = queue.poll();

int r = cell[0], c = cell[1];

for(int[] dir : dirs) {

int x = r + dir[0], y = c + dir[1];

if ( x < 0 || x >= m ||

y < 0 || y >= n ||

rooms[x][y] != Integer.MAX\_VALUE

) continue;

rooms[x][y] = rooms[r][c] + 1;

queue.add(new int[] {x,y});

}

}

}

1. **Test**