286. Walls and Gates Medium Breadth-First Search Array Matrix **Leetcode Link** 

## **Problem Description**

blocked wall. The value -1 identifies a wall or obstacle, which means the room cannot be passed through. A value of 0 indicates a gate, serving as a possible destination for other rooms. Any room with a value of INF, which stands for infinity and is given the numerical value of 2147483647, represents an empty room needing the distance filled to its nearest gate.

In this problem, we are presented with a grid representing a series of rooms, each of which could be an empty room, a gate, or a

The task is to update the grid so that all empty rooms have their values changed from INF to the shortest distance to a gate. If there's no way to get to a gate, the value of INF should remain unchanged. The update should be done in place, meaning no additional grid should be constructed but instead, the rooms grid itself should be modified.

#### The solution applies the Breadth-First Search (BFS) algorithm, a common approach for exploring all possible paths in level-order from a starting point in a graph or grid.

Intuition

The intuition for this problem is as follows: 1. First, we identify all gate locations as starting points since we need to find the minimum distance from these gates to each room.

We continue this process until there are no more rooms to explore.

- 2. We put all these gate coordinates in a queue for BFS processing. Since BFS processes elements level-by-level, it's perfect for measuring increasing distances from a starting point (in this case, the gates).
- 3. We pop coordinates from the queue and explore its neighbors (rooms to the left, right, up, and down). If we find an empty room (with INF), it's apparent that this is the first time we reach this room (as the queue ensures the shortest paths are explored first), so we update the room's value to the current distance.
- 4. Since gates serve as the origin, distances increase as we move away from them. Each level deeper in the search increases the distance by one.

5. Neighbors that are walls or already visited with a smaller distance are skipped, as we are looking for the shortest path to a gate.

- This approach ensures that each empty room is attributed the shortest distance to the nearest gate by leveraging the level-order traversal characteristic of the BFS, which naturally finds the shortest path in an unweighted grid.
- **Solution Approach**

iteratively. The queue data structure is chosen for its ability to handle FIFO (First-In-First-Out) operations, which is essential for BFS. Here is a step-by-step breakdown of the implementation:

The solution leverages the Breadth-First Search (BFS) algorithm, utilizing a queue to process each gate and its surrounding rooms

## queue. This is done because gates are our BFS starting points.

2. **BFS Implementation**:

A while loop commences, indicating that we continue to process until the queue of gates and accessible rooms is empty.

• A nested for loop enables us to process rooms level-by-level. It iterates through the number of elements in the queue at the

1. Initialization: The rooms grid is scanned for all gates (0 value rooms). Their coordinates are added to a deque, a double-ended

start of each level, to separate distances incrementally. Each room's coordinates are popped from the deque, and for each room, we check its four adjacent rooms (top, bottom, left,

for the next level of rooms.

**Example Walkthrough** 

right). 3. Neighbor Checks: For every neighboring room, if the neighbor coordinates are within the bounds of the grid and the neighbor is an empty room (INF value), it means we've found the shortest path to that room from a gate. Therefore:

• The empty room is updated with the distance d, representing the number of levels from the gates we've traversed.

4. Distance Increment: Once we've explored all rooms from the current level, we increment d by 1 to reflect the increased distance

The algorithm concludes when there are no more rooms to explore (the queue is empty), ensuring all reachable empty rooms have

been filled with the shortest distance to a gate, and non-reachable rooms are left as INF.

This room's coordinates are then added to the queue for subsequent exploration of its neighbors.

avoiding the creation of additional data structures and ensuring optimal space complexity.

Let's illustrate the solution approach using a simple 3×3 grid example where −1 represents a wall, 0 represents a gate, and INF

By using this approach, the solution efficiently updates the rooms with their minimum distances to the nearest gate in place,

(represented as 2147483647 for the purpose of example) represents an empty room that needs its distance filled to the nearest gate. Consider the following grid:

Here's how the Breadth-First Search (BFS) algorithm would update this grid:

## We identify the gate at grid[0][2] and add its coordinates to the queue.

**BFS Implementation:** 

INF -1 INF

Initialization:

**Neighbor Checks:** 

#### 1 INF -1 0 INF -1 INF

**Distance Increment:** 

The grid now looks like this:

Increment d to 2 for the next level of rooms.

(1, 1) is updatable and becomes 2.

(2, 2) is updatable and becomes 2.

• For (1, 1), neighbors are (1, 0), (1, 2), (0, 1), and (2, 1).

• (1, 0) is updatable and becomes 3.

(2, 1) is updatable and becomes 3.

Start with the gate in the queue: (0, 2).

• The gate at (0, 2) has three neighbors: (0, 1), (0, 3), and (1, 2).

○ However, (0, 1) is a wall and (0, 3) is out of bounds, so we only consider (1, 2).

Now, (1, 2) is the only room in the queue. Its neighbors are (1, 1), (1, 3), (0, 2), and (2, 2).

• Since (1, 2) is INF, we update it to the distance 1 (the current level), and add (1, 2) to the queue.

#### • (1, 3) is out of bounds. o (0, 2) is a gate and already has the shortest distance.

3 INF -1 2

Continue BFS:

**Distance Increment:** 

 Increment d to 3. **Next BFS level:** 

Add (1, 1) and (2, 2) to the queue.

The grid now looks like this:

For (2, 2), the only neighbor not already checked or out of bounds is (2, 1), but it's already been updated this level.

Final updated grid:

### Notice that the top left room remains INF, as it is inaccessible. All other rooms have been updated to show the shortest path to the nearest gate.

**Python Solution** 

class Solution:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

34

35

36

37

38

39

40

41

42

43

44

41

42

43

44

45

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

33

34

35

36

37

38

C++ Solution

public:

1 class Solution {

void wallsAndGates(vector<vector<int>>& rooms) {

for (int row = 0; row < rows; ++row) {</pre>

// Distance from gate(s)

// Use BFS to traverse the grid

while (!toVisit.empty()) {

int distance = 0;

++distance;

// Find all gates and enqueue their positions

for (int col = 0; col < cols; ++col) {</pre>

toVisit.emplace(row, col);

**if** (rooms[row][col] == 0) {

size t levelSize = toVisit.size();

for (size\_t i = 0; i < levelSize; ++i) {</pre>

toVisit.pop(); // remove it from the queue

int rows = rooms.size(); // number of rows in the grid

int cols = rooms[0].size(); // number of columns in the grid

queue<pair<int, int>> toVisit; // queue to maintain BFS traversal from gates

vector<int> directions =  $\{-1, 0, 1, 0, -1\}$ ; // directions for exploring adjacent rooms

auto position = toVisit.front(); // get the front element of the queue

// Check for valid position and if the room has not been visited or is not a wall

rooms[newRow][newCol] = distance; // set distance from nearest gate

toVisit.emplace(newRow, newCol); // enqueue the valid position

if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && rooms[newRow][newCol] == INT\_MAX) {</pre>

for (int j = 0; j < 4; ++j) { // explore all 4 adjacent rooms

int newRow = position.first + directions[j];

int newCol = position.second + directions[j + 1];

from collections import deque

INF = 2\*\*31 - 1

def wallsAndGates(self, rooms):

:type rooms: List[List[int]]

# Dimensions of the rooms matrix

i, j = queue.popleft()

num\_rows, num\_cols = len(rooms), len(rooms[0])

# Define the representation of an infinite distance (empty room)

# Initialize a queue and populate it with the coordinates of all gates

# Explore the four possible directions from the current cell

new\_row, new\_col = i + delta\_row, j + delta\_col

# Update the distance for the room

rooms[new\_row][new\_col] = distance

queue.append((new\_row, new\_col))

for delta\_row, delta\_col in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

# Check if the new position is within bounds and is an empty room

# Add the new position to the queue to process its neighbors

22 # Initialize distance from gates 23 distance = 0 24 # Perform a breadth-first search (BFS) from the gates 25 26 while queue: # Increase the distance with each level of BFS

if 0 <= new\_row < num\_rows and 0 <= new\_col < num\_cols and rooms[new\_row][new\_col] == INF:</pre>

queue = deque([(row, col) for row in range(num\_rows) for col in range(num\_cols) if rooms[row][col] == 0])

This method modifies the 'rooms' matrix in-place by filling each empty room with the distance to its nearest gate.

An empty room is represented by the integer 2\*\*31 - 1, a gate is represented by 0, and a wall is represented by -1.

```
27
28
                distance += 1
29
30
                # Process nodes in the current level
31
                for _ in range(len(queue)):
32
33
```

```
45
Java Solution
  1 class Solution {
         public void wallsAndGates(int[][] rooms) {
             // get the number of rows and columns in the rooms grid
             int numRows = rooms.length;
             int numCols = rooms[0].length;
  6
             // create a queue to hold the gate positions
             Deque<int[]> queue = new LinkedList<>();
  8
  9
             // find all gates (represented by 0) and add their positions to the queue
 10
             for (int i = 0; i < numRows; ++i) {</pre>
 11
 12
                 for (int j = 0; j < numCols; ++j) {
 13
                     if (rooms[i][j] == 0) {
 14
                         queue.offer(new int[] {i, j});
 15
 16
 17
 18
 19
             // distance from the gate
 20
             int distance = 0;
 21
             // array to facilitate exploration in 4 directions: up, right, down, left
 22
             int[] directions = \{-1, 0, 1, 0, -1\};
 23
 24
             // perform BFS starting from each gate
 25
             while (!queue.isEmpty()) {
 26
                 ++distance;
 27
                 for (int i = queue.size(); i > 0; --i) {
 28
                     // get and remove the position from the front of the queue
                     int[] position = queue.poll();
 29
 30
                     for (int j = 0; j < 4; ++j) { // explore all 4 directions
 31
                          int newRow = position[0] + directions[j];
 32
                         int newCol = position[1] + directions[j + 1];
 33
                         // if the new position is within bounds and is an empty room
                         // (denoted by Integer.MAX_VALUE), then update the distance
 34
                         // and add the new position to the queue for further BFS
 35
 36
                         if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols && rooms[newRow][newCol] == Integer.MAX_</pre>
 37
                              rooms[newRow][newCol] = distance;
 38
                              queue.offer(new int[] {newRow, newCol});
 39
 40
```

#### 27 28 29 30 31 32

```
39
 40
 41
 42
    };
 43
Typescript Solution
    function wallsAndGates(rooms: number[][]): void {
         const rows = rooms.length; // number of rows in the grid
         const cols = rooms[0].length; // number of columns in the grid
         const toVisit: Array<[number, number]> = []; // queue to maintain BFS traversal from gates
         const directions: number[] = [-1, 0, 1, 0, -1]; // directions for exploring adjacent rooms
  6
         // Find all gates and enqueue their positions
         for (let row = 0; row < rows; row++) {</pre>
  8
             for (let col = 0; col < cols; col++) {</pre>
  9
 10
                 if (rooms[row][col] === 0) {
 11
                     toVisit.push([row, col]);
 12
 13
 14
 15
 16
         // Distance from gate(s)
         let distance = 0;
 18
 19
         // Use BFS to traverse the grid
         while (toVisit.length > 0) -
 20
 21
             const levelSize = toVisit.length;
 22
             distance++;
 23
             for (let i = 0; i < levelSize; i++) {</pre>
 24
                 const position = toVisit.shift(); // get the front element of the queue
 25
                 if (position) {
                     const [currentRow, currentCol] = position;
 26
 27
 28
                     // Explore all 4 adjacent rooms
 29
                     for (let j = 0; j < 4; j++) {
                         const newRow = currentRow + directions[j];
 30
 31
                         const newCol = currentCol + directions[j + 1];
 32
 33
                         // Check for valid position and if the room has not been visited or is not a wall
 34
                         if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && rooms[newRow][newCol] === Number.MAX_SAFE_I</pre>
 35
                              rooms[newRow][newCol] = distance; // set the distance from the nearest gate
 36
                              toVisit.push([newRow, newCol]); // enqueue the valid position
 37
```

# Time and Space Complexity

#### rooms matrix. This is because in the worst case, the code must visit each cell in the matrix once. Starting from each gate (where the value is 0), the algorithm performs a breadth-first search (BFS), updating distances to each room that is initially set to inf. Every room is pushed to and popped from the queue at most once. The BFS ensures that each room is visited only when we can potentially

Time Complexity

38

39

40

41

42

43

**Space Complexity** The space complexity of the given code is also 0(m \* n), since in the worst case, we could have a queue that contains all the cells (in the case where all rooms are reachable and no walls are present to block the spread). This queue is the dominant factor in the space complexity of the algorithm, as it could potentially hold a number of elements equal to the total number of rooms at once. The additional space used by the BFS for coordinates and related computations is negligible compared to the size of the queue.

The time complexity of the given code is 0(m \* n), where m represents the number of rows and n the number of columns in the

write a smaller distance. Hence, the time complexity is linear in the size of the input matrix.